PERICLES - Promoting and Enhancing Reuse of Information throughout the Content Lifecycle taking account of Evolving Semantics

[Digital Preservation]

DELIVERABLE 6.4

Final version of integration framework and API implementation

GRANT AGREEMENT: 601138

SCHEME FP7 ICT 2011.4.3

Start date of project: 1 February 2013

Duration: 48 months

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination level		
PU	PUBLIC	X
PP	Restricted to other PROGRAMME PARTICIPANTS (including the Commission Services)	
RE	RESTRICTED to a group specified by the consortium (including the Commission Services)	
CO	CONFIDENTIAL only for members of the consortium (including the Commission Services)	

Revision History

V #	Date	Description / Reason of change	Author
V0.1	17/08/2015	Initial Skeleton and Description Work	AG (UEDIN)
V0.2	09/09/2015	Initial Draft Near Completion	AG (UEDIN)
V0.3	11/09/2015	Restructured Draft and Started Language Harmonisation	AG (UEDIN)
V0.4	07/10/2015	Redraft and expansion based upon internal review comments	AG (UEDIN)
V0.5	22/10/2015	Internal Review and Restructuring, Expansion of Client Applications	AG (UEDIN)
V0.6	23/10/2015	Main Reworking of Client applications done	AG (UEDIN)
v0.7	23/10/2015	Final editorial pass	RMB (UEDIN)
v1.0	30/10/2015	Final version	CS, MH, SW (KCL)

Authors and Contributors

Contributors at this WP are the end user partners. WP1 provides feedback to technical partners.

Authors

Partner	Name
UEDIN (lead)	Alistair Grant (AG), Charaka Palansuriya (CP), Rob Baxter (RMB)
UGOE	Noa Campos López (NC), Marcel Hellkamp (MH)
DOTSOFT	George Antoniadis (GA), Tania Petsouka (TP)
SPACEAPPS	Emanuele Milani (EM)
KCL	Emma Tonkin (ET), Simon Waddington (SW)
ULIV	Jerome Fuselier (JF)

Contributors/Reviewers

Partner	Name
UGOE	Johannes Biermann (JB)
CERTH	Stratos Kontopoulos (SK)

Table of Contents

Glossary

Abbreviation / Acronym	Meaning
BPMN	Business Process Model and Notation
Docker	Container Based Application Virtualisation Software
ERMR	Entity Registry Model Repository
GIT	Distributed Version Control System
Handler	Web Service (REST based) Communications Component
jBPM	Java Implementation Suite for BPMN
Jenkins	Continuous Integration Software Tool
JSON	Javascript Object Notation
LRM	Linked Resource Model
LRMS	Linked Resource Model Service
Maven	Build Automation Tool used primarily for Java
MICE	Model Impact Change Explorer
PC	Process Compiler
PE	Processing Element
REST	Representational State Transfer
TMS	The Museum System
UAG	User Advisory Group
UC	Use Cases
WE	Workflow Engine
xIP	Type X Information Package
xUnit	Test Frameworks for a range of languages
YAML	a human friendly data serialization standard for all programming languages (quote from http://yaml.org/)

1 Executive Summary

The PERICLES integration framework is designed for the flexible execution of varied and varying processing and control components in typical preservation workflows, while itself being controllable by abstract models of the overall preservation system. It is the project's focal point for connecting tools, models and application use-cases to demonstrate the potential of model-driven digital preservation.

This final design for the integration framework has changed slightly from the initial version presented in PERICLES deliverable D6.1 [10]. We describe the changes and the reasons for them in the early chapters of this report.

The integration framework is built from standard encapsulation technologies – Docker containers and RESTful web services – and controlled by a standard workflow environment – jBPM controlled by the Jenkins continuous integration system. On this execution layer, arbitrary workflows representing digital preservation activities can be deployed, run and evaluated. Standard tools – mediainfo, bagit, fido and so on – can be encapsulated and deployed, as can new preservation tools developed within the project.

Two new subsystems have been designed to couple the workflow execution layer to the abstract models developed through the research activities of the project: the Process Compiler (PC) and the Entity Registry-Model Repository (ERMR). The ERMR also provides the key link to the Linked Resource Model Service, an external semantic reasoning service under development by partner XEROX Research. These two subsystems provide the means to couple powerful semantic reasoning and policy-driven models to a "live" digital preservation system.

The API designs and technology choices for the test bed are now settled and implementation of the underlying (standard) test bed infrastructure is complete. The APIs and communication patterns are based on RESTful web services and JSON payloads and are described in detail in Section 5 and in the appendices.

Implementation of the new ERMR and PC subsystems is well underway. The focus for the integrated test bed over the final stages of the project will be on demonstrating the full end-to-end power of the model-driven preservation approach through the implementation of key application scenarios using models, tools and components drawn from across the PERICLES project. Examples of such scenarios are given in the appendices.

2 Introduction & Rationale

2.1 Context of this Deliverable

In exploring what we term model-driven preservation, PERICLES has been developing and extending technologies and concepts to cope with the changes that would be required over the lifetime of complex digital objects, as well as the underlying preservation environment itself, in order to maintain reusability. To accommodate this wide range of potential targets, scenarios focussed around small segments of functionality have been developed which will show how the PERICLES approach will operate both within subsets of an archive and over the lifetime of the digital objects themselves. This document describes the final version of the integration framework developed to support this scenario-based approach to archive and preservation operations and facilitate easier testability and usability evaluation.

PERICLES is about developing techniques and concepts for building long life preservation systems; the integration framework will support the testing and demonstration of this goal.

2.1.1 Relation to other work packages

WP6 takes on the primary responsibility for ensuring that the integration framework and underlying test bed is operational and that the functionality of the test bed is sufficient to support the project goals. Other work packages interact with WP6 in order to make best use of the framework – essentially, showing how the different concepts and technologies in the project will link together to support the project goals.

WP2 describes user scenarios in concert with support from the other work packages – a large part of this interaction is focussed around ensuring that the scenarios and tests on the test bed make sense from a specialist domain view point. If the test scenarios do not have a strong relationship to the expert reality then the results from them will be unusable.

WP3-5 uses the framework to help show how their technology and concepts support not only the user scenarios but also wider ranging and possibly more abstract scenarios developed in the research-oriented elements of the project. This involves capturing the concepts into coherent test plans and providing accurate and timely software and environment requirements to the developers in WP6 who assist them in deploying scenarios on the test bed and provide advice on how the framework supports their envisioned component structures.

2.2 What to Expect from this Document

Deliverable 6.4 describes the framework for testing and integration of PERICLES components and concepts. This document describes the framework purpose, how it will fulfil this purpose and how the framework would influence the development of a real world archive based on PERICLES work. To support and illustrate these aims, the appendices describe a number of preservation scenarios which have been developed in the course of the project.

2.3 Document Structure

The structure of this document follows the general deliverable template agreed by the Consortium. The main purpose of the deliverable is to:

- Describe the integration and test framework. Chapter 3 gives an overview of the framework including changes from prior deliverables, which have been necessary to accommodate the project requirements.
- Motivate how the framework can be used to inform how real-world implementations of a PERICLES system could be created. Chapter 6 will introduce the similarities and differences from the test bed to a real world system, highlighting the major issues, which are likely to occur in such a development.
- Detail and list the required communication types and methods for the components within PERICLES. Chapter 5 presents an overview of the communications methods and protocols that are to be used in the integration and test bed frameworks.
- Detail the Architecture upon which the framework is based. Chapter 4 is a description of the major components in a PERICLES system and how they fit into the test bed.

The appendices contain supporting material to illustrate the concepts and technology that the framework has to enable:

- Scenario Descriptions – Policy Change and Noise Reduction
- Supporting Technical Documentation on Communications

3 The PERICLES Integration Framework

3.1 Integration Framework Goals in PERICLES

The integration framework and its deployment across the project test beds supports three main goals in PERICLES: reproducible testing, component reconfiguration and unified integration approach.

The most compelling purpose of the integration framework is to unify the way in which the concepts and components developed in PERICLES are combined and tested:

- Components need to be packaged in a coherent format for testing – through the use of automated scripting
- Tests should adhere to the same standards
- Reporting is generated in a standard fashion and accessible at a singular location for ease of use

Reproducible testing supports reliable research carried out in the test bed and should allow results and behaviours to be replicated by others using the integration framework in their own testing. In this project, reproducible testing in the framework encompasses the following:

- Tests which include multiple components operating across different data types, operating systems, software types and user scenarios
- Tests that confine side effects and artefact generation to their own test environments – no test should adversely affect the running of another test
- Documented and automated setup and configuration of test runs as supported by a managed test environment
- Self-documenting results for ease of interpretation

Component reconfiguration relates to standard testing practices where individual components can be configured to support testing of different approaches and scenarios. For PERICLES, this supports the following:

- Simulation of real world changes in the test environment to allow testing of system behaviours and component reactions
- Flexibility, allowing scenario and test implementers to use a core set of components in different ways through automated configuration rather than complete multiple versions of preconfigured software

In implementing the integration framework and test beds against these goals we have drawn on recent work on *software in research* being carried out by the Software Sustainability Institute [13].

3.1.1 Integration Framework: what is it for?

The integration framework and test bed within PERICLES supports four key areas of application: scenario enactment, technology testing, concept demonstration and feedback incorporation. These areas are, of course, intertwined within the test bed.

Scenario enactment is the principal driver of how testing is done within PERICLES. The application partners develop scenarios based around their domain knowledge and practices that could be affected, improved or further developed through the use of the concepts and technologies created within PERICLES. The framework allows for these scenarios to be developed into multi-stage processes, which can then test and examine the whole scenario (such as the application of policy

change in section A) or parts of a scenario in isolation. This enables PERICLES to examine its developments in 'real world' situations determined by experts in their fields of preservation.

Technology testing, though facilitated by scenario enactment, can be considered separate, as often a piece of technology will need to be tested in isolation in different configurations and patterns before integration into a user scenario. This type of testing can include how to setup server software in different patterns (single host, multi-host, cloud patterning), or to how to configure best the use of multiple packages on a single machine to be exposed as a single technical service for use in a scenario. PERICLES encompasses both new software developments in the project and third party software in its work and as such needs to have a way to evaluate how best to deploy and employ these components. To support this approach, the framework employs web service handlers (Section 4.2.5) and container-based virtualisation.

Concept demonstration is where newly developed concepts in PERICLES that may or may not be directly derived from user requirements can be shown. Much of the work on the model-driven approach has to be shown to work on a conceptual level before integration into real world scenarios. Thus the framework needs to support these 'what-if?' scenarios, which allow the developers of the model-driven approach to demonstrate the concepts in a clean and unencumbered fashion with simulated data and stimuli.

Feedback incorporation is a key tenant of all software testing – don't test unless you are going to make use of the results. There is little point in testing whether or not a component or scenario works if, when it fails or doesn't behave as expected, you ignore this and carry on regardless. The feedback from the tests allows the concepts and components to be improved and rectified.

3.1.2 What the Framework and Test Bed are not

It is important to state that the test bed and framework are not being used for full archive implementation in the PERICLES project.

The test bed and framework are used to show snapshots and subsystems of PERICLES technology and concepts in a working environment; it is not within our scope to recreate a full working archive. Rather than create another digital archive with a necessarily rather limited horizon, PERICLES's aim is to showcase and provide proof of concept for preservation research ideas. Section 6 introduces ideas on translating PERICLES concepts from test bed to real world implementation.

3.2 Using the Integration Framework

3.2.1 Scenario summary

For testing and scenario demonstrations, the PERICLES project follows a set of guidelines that all such scenarios are based around. These guidelines are to ensure that all test cases and demonstrations are developed in a unified process for compatibility, maintenance and ease of understanding.

Each scenario is described as a *test case* with the following items:

- Scenario description
- Required technology
- Test/sample data
- Process
- Evaluation/success criteria

Scenario descriptions should contain a concise summary of what the scenario is intended to demonstrate or test – this can be a scenario which is derived from the application use cases

developed by WP2 or technology or concept scenarios developed by WP3-5. In many cases it will likely be a combination. The description should summarise the aim, the broad process and main criteria for analysis.

Required technology should describe or list the software stack needed to perform the operations described in the scenario. This should include all software that needs to be employed by the workflow engine during the operation of the process, and any analysis tools required by the driving test code.

Test and sample data includes any data required for the test or scenario to run, including input and expected output data when appropriate. The output may not be in a form readily testable, in which case the scenario developers must supply a method by which the test runner can verify if the test has run and the method should provide a metric (simplest would be pass/fail output) about the scenario as to whether or not it was successful. The data should not rely on having been processed by any other scenario in the test bed. This isolation is important at this stage.

Process should detail all the steps required to carry out the scenario in all its potential paths – these paths should be detailed in the test plan as separate cases under the one description. The process should include the pre-run setup of test data, required BPMN processes and technology installations, concluding with post-run analysis and reporting.

Evaluation/success criteria define how the outputs of the scenario under operation are interpreted. For many tests and demonstrations it will be a simple pass/fail metric where outputs are measured against a set of criteria determined by the scenario developers. Some scenarios may require more complex analysis or require interpretation of the results by a person. The test description needs to define this process. All tests and scenarios should include a mechanism to indicate its status.

3.2.2 Reporting mechanism

The test bed uses Jenkins[1] to manage its running of test cases and scenarios. As such all test outputs are reported back via the Jenkins test dashboard. This dashboard uses simple colour coding to indicate pass/failure in the test suites. It gives additional information about stability and allows further probing of test cases via its internal links. This is the central access point for the project to view the status of the current scenarios using the integration framework.

[1] Jenkins continuous integration tool. See https://jenkins-ci.org/

4 Test Bed Integration

4.1 Design Description

Deliverable D6.1 [10] described the initial design of the test bed. Over the course of the last months of detailed design this has been refined, with one major change made to the architecture. This subsection will update the current design of the test bed. It updates component descriptions from Deliverable D6.1 where appropriate, and give small component summaries.

The original architecture for the test bed and integration framework from Deliverable D6.1 is displayed in Figure 4-1. Since the publication of D6.1, the architecture has changed in parts to reflect a better understanding of the framework requirements and the implementation factors that influence its development.

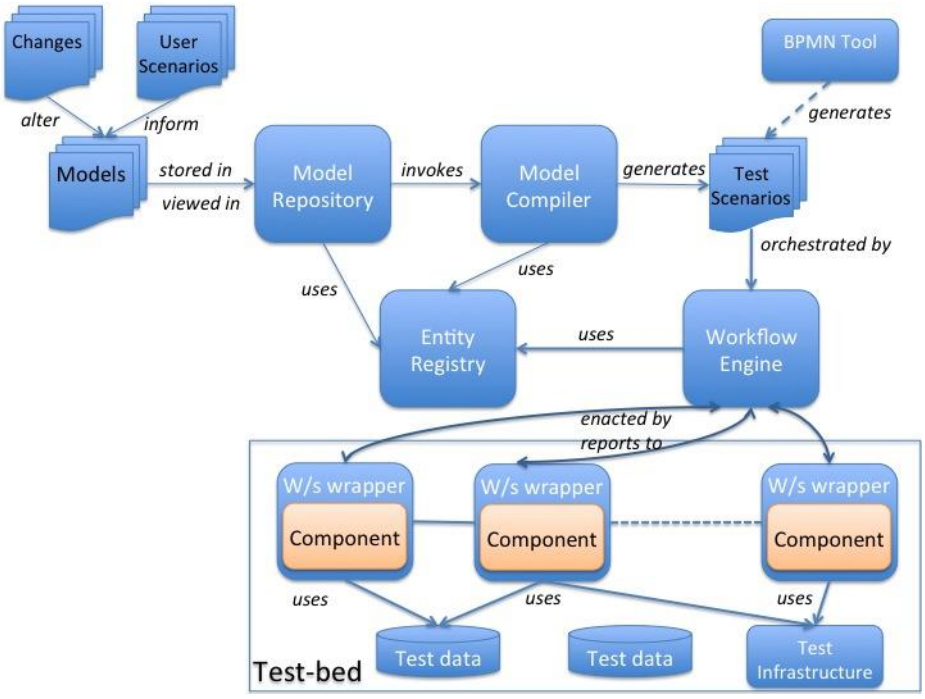

Figure 4-1 Original Test-Bed Architecture (D6.1)

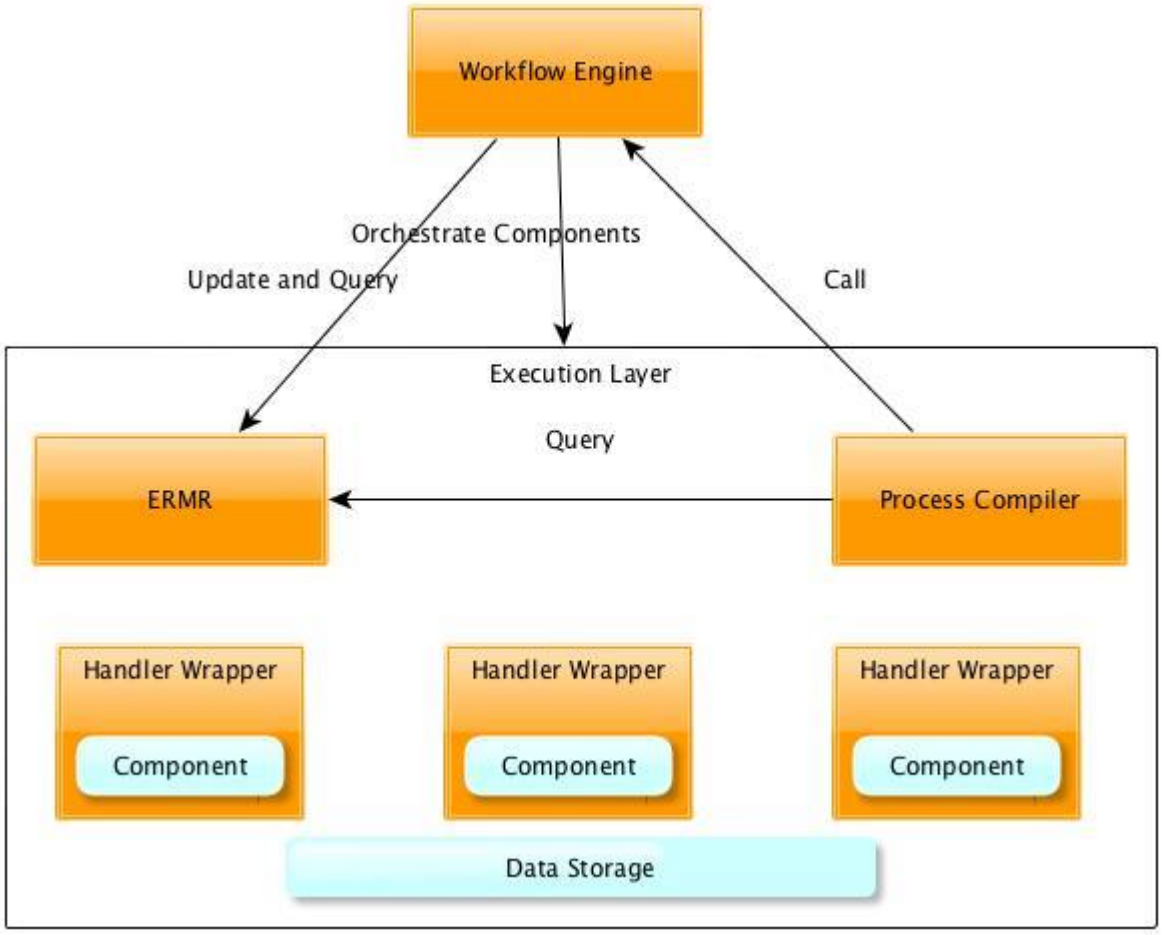

Figure 4-2 Final Test-Bed Architecture

The major change to the design from D6.1 to current is the merging of the *Entity Registry* and *Model Repository* into a single *ERMR* component as shown in Figure 4-2 Final Test-Bed Architecture. The design team decided that the crossover in functionality and the level of interconnection required for good operation was large enough that merging the two components made sense and gave a more robust architecture on which to proceed. This merges the prior component purposes resulting in a single component for registering entities and models.

As part of this change, the ERMR includes a *mediator* function, which allows the reasoning capabilities of the Linked Resource Model Services (LRMS) to be accessed through the ERMR. This puts the LRM Service logically behind the ERMR, allowing the querying mechanisms to be exposed at a singular interface and as such reduce the dependency in the overall architecture on the LRM Service (LRMS). This not only integrates the LRMS as a final component in the test bed architecture, but does so in a way that the reasoning service can be updated and altered without major impact upon the workflows and process models assuming that the changes do not alter the external behaviours of the reasoning system.

Figure 4-2 Final Test-Bed Architecture represents how the test bed will implement a PERICLES process: in the test bed every component (other than the workflow engine) is an instantiation on an execution layer (which is where all components are being installed and executed). The workflow engine will take a test scenario and instantiate all the components that will be present in a PERICLES system – including populating test data. This differs from how a real world production version of a

PERICLES preservation management system might be deployed (Figure 4-3 Real World Architecture). The main difference here is that there are additional fixed points in this system: in particular the ERMR becomes a fixed point, as does the data storage. Thus a PERICLES system has three main fixed points – this could vary depending on implementation but would not go below three as this is the minimum coordinating and data storage functionality required. It should be noted that the Process Compiler is not a fixed point in the system and can exist on the Execution Layer as a deployable execution unit.

By "fixed point" here we mean a subsystem that is required to control the rest of the test bed execution layer and thus is "always on". Other subsystems can be deployed and removed from the test bed by the controlling fixed-point subsystems. In the project's use of the test bed, everything can be deployed or removed by the workflow engine alone. In a "production" system, the ERMR and the Process Compiler become key subsystems for the management of the preservation system realised on the execution layer; thus, they become fixed points in the overall system.

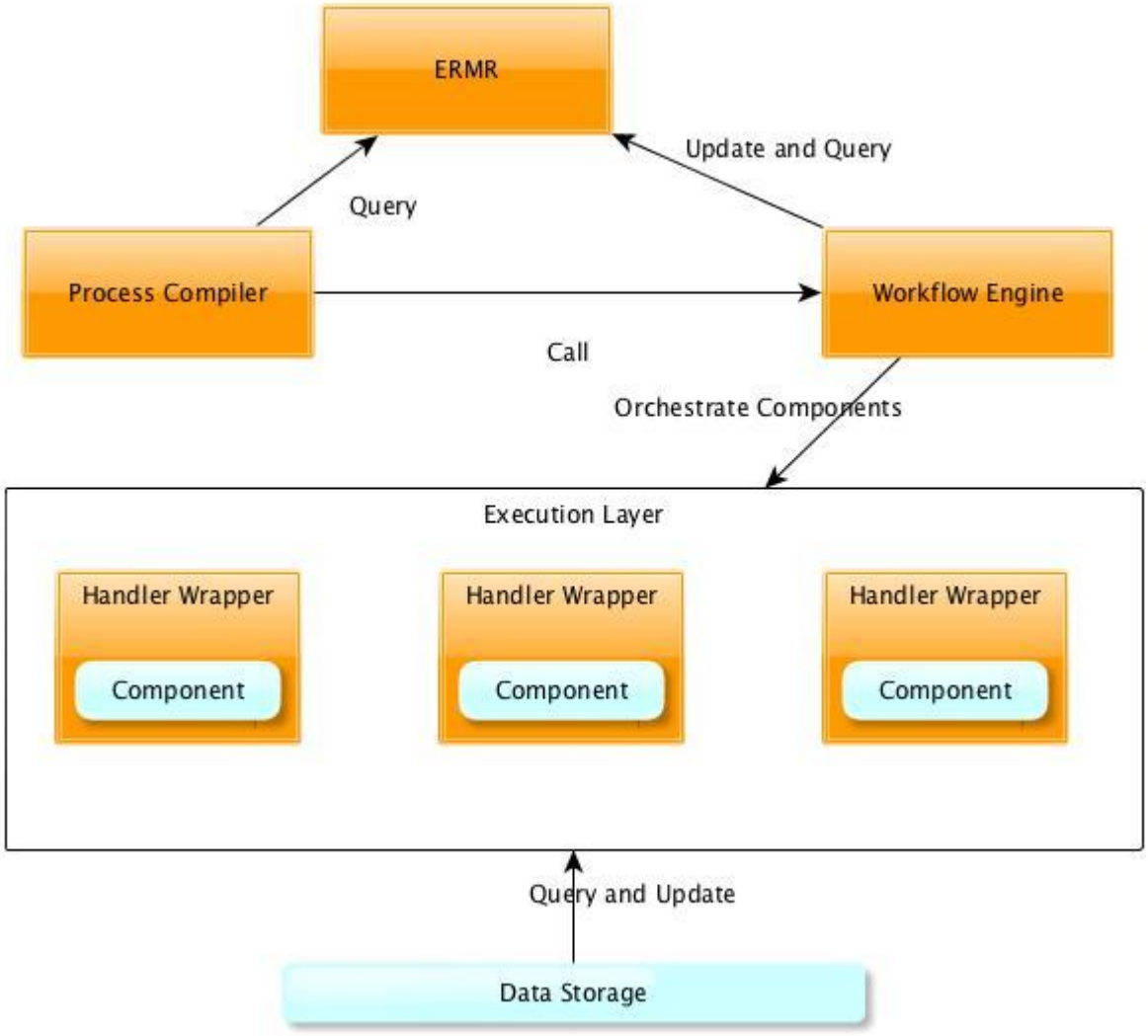

Figure 4-3 Real World Architecture

4.2 Major Components

4.2.1 Entity Registry Model Repository

The Entity Registry Model Repository (ERMR) is a component for the management of digital entities and relationships between them. Access methods are presented as RESTful services (see Section 5.2).

The ERMR is used to store and register entities and models. Agreed metadata conventions will be used to provide the necessary registry functionality; the registry is agnostic to the entities and metadata stored with the interpretation of data being the responsibility of the client applications. The ERMR provides a CDMI implementation[2] for HTTP access to entities in the registry.

The ERMR uses a triple store as a database for the storage and retrieval of triples through semantic queries. The ERMR also provides a mediator service to integrate semantic services (like the LRMS) to extend its reasoning capabilities (as shown in the diagram Figure 4-4). The triple store provides a simple RESTful API for access to the registry. Queries can be expressed in the SPARQL query language to retrieve and manipulate data stored in the triple store. To link entities described as triples to actual digital objects, the ERMR is able to provide unique identifiers used to create a unique CDMI URL for an associated object. This URL can be used in a triple to link entities and digital objects stored in a data storage component.

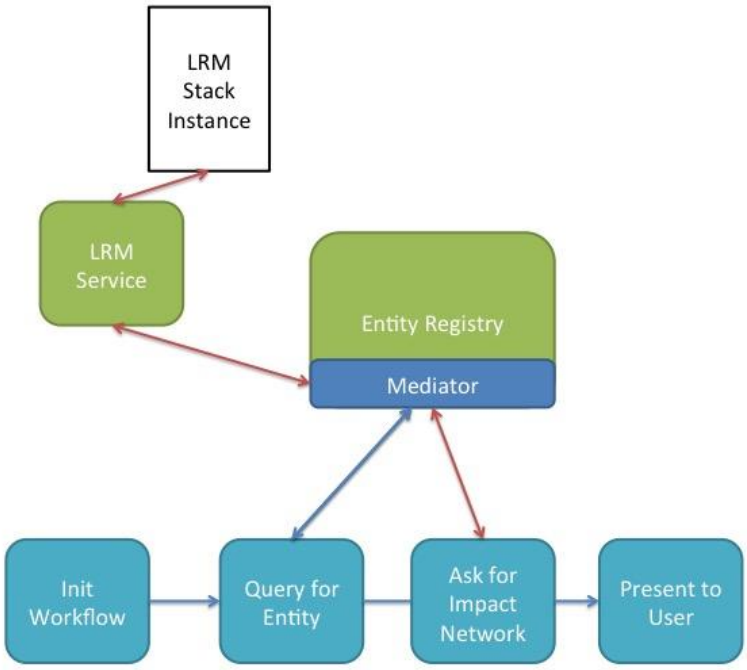

Figure 4-5 Mediator and ERMR

Figure 4-5 shows a workflow that initially queries the ERMR; this query is intercepted by the mediator function that decides if the query should be handled by the ERMR itself or passed to the LRMS to make use of the semantic reasoning services. In the diagram, the blue lines are

[2] Cloud Data Management Interface, a storage industry standard for RESTful access to distributed storage. See http://www.snia.org/cdmi

communications dealt with by the ERMR; the red lines show communications that are passed to the LRMS. It is important to note that to the workflow this is a singular interface, it does not need to understand the structure behind the interface.

4.2.2 Workflow Engine

The Workflow Engine (WE) takes processes, compiled by the Process Compiler with descriptions and implementations stored in the ERMR, Data Storage and Workflow Engine cache, and executes them to orchestrate executable components (PERICLES tools, archive subsystems or supporting software) wrapped in Web Services Handlers with REST targets. This is one of the main fixed points of the architecture, which is active at all times. It orchestrates and run the processes described in BPMN [2], which themselves are implementations of the concepts and operations required by the relevant scenarios. The workflow engine is the central contact point for any operation initiated within a "PERICLES system", where workflows (*processes* in PERICLES terms) can be run concurrently with the ability to spawn new processes as required.

The Workflow Engine supports:

- concurrent workflow execution
- parameterised workflow execution
- REST target and communications
- user and service level security
- business rule decisions.

This component communicates with all other major components, both to launch operations and have operations launched from them (communication patterns are covered in detail in Section 5). This component has not changed since D6.1 in its function.

4.2.3 Processing Elements

Processing Elements (PEs) are any pieces of software or hardware that can be used by a PERICLES process to accomplish a given task. Each Processing Element is described by the following:

- *Name* – Common or actual name for software/hardware being used.
- *Identifier* – system assigned identifier.
- *Function Description* – description of what functions that the processor provides in terms defined by the system administrators.
- *Version* – version of this instance of the software/hardware.
- *Platform* – where appropriate the platform the processor requires for operation.
- *Input Type(s)* – what are the inputs to the processor in terms of ordering, types, and whether they are optional or not.
- *Output Type(s)* – what are the outputs from the processor in terms of ordering, types, and whether they are optional or not.
- *Known Issues* – are there any current or past issues with the processor that need to be recorded – for example certain combinations of input and output types cause erroneous behaviour.
- *Deployment Type* – whether the processor is a permanent installation, deployable service or a third-party service.

Within PERICLES, these Processing Elements will be operators, to which the Process Entities refer when populating PERICLES processes in the Process Compiler and Workflow Engines.

4.2.4 Data Storage

Data Storage is a specialised long-term Processing Element, a permanent service available to a PERICLES system. This component is responsible for storing *digital objects*, which can be data files, metadata, models or ontologies.

Data Storage must be represented in the PERICLES system as a long-term service, since PEs are typically transient in nature with limited functionality scope. The Data Storage service must manage data, as required, as bit level preservation, object replication and distributed storage mechanisms.

It should be noted that, while the ERMR could in theory provide a data storage element, that is not its purpose within the framework described; the Data Storage element will manage the storage requirements of an archive developed using this framework with the ERMR being used for metadata and modelling functions.

Other functionalities Data Storage can provide include file fixity checking, version control or quarantine, although these might also be realised using other Processing Elements within a workflow, as dictated by policies applied to a given PERICLES system.

4.2.5 Web service component wrappers (Handlers)

The *Handlers* are the core of the integration framework in PERICLES. They function as the communication points for each major entity within the system. The Handlers deal with the validation of incoming requests, exercise the functionality of the PEs they wrap, store and transfer the results of PE functions and initiate necessary communications with other PEs.

The Handlers do not perform any operations, which change or alter the data contained in the objects they handle; only PEs can alter and change data. PEs, on the other hand, should not have knowledge of anything in their environment – PEs behave as highly functional but 'dumb' pieces of software. This means they perform their function and only their function, and the Handlers deal with the rest of the PERICLES system and the outside world.

This separation of concerns simplifies the job of the PE developers – they only have to provide a method, be it a shell script, executable or callable function, to the Handler which will be used to exercise the functionality.

Additional functions can be added to the Handler based on the needs of a particular PE; these are designed to be plugins to the Handler and are not mandatory. These functions could include, for instance, data validation: was the right data format passed to the handler for the PE to operate on?

This component has not changed since D6.1 in its function.

4.2.6 Process Compiler

Formerly known as the Model Compiler, detailed design work indicated that this component be renamed Process Compiler (PC) as the component is designed to take one representation of a process model and transform it into another form that can be executed by the workflow engine. As part of this the PC will transfer information to the ERMR, which is used to update the process descriptions. The current component implementation is targeted towards a BPMN compilation system though in theory the design can be adapted to any workflow engine language.

The main functionality of the PC is to support the translation and reconfiguration of preservation process models described in the Entity Registry Model Repository (ERMR) into executable workflows to be employed by the Workflow Engine.

As preservation processes are part of the ecosystem they are modelled and described in the ERMR via the LRM (Linked Resource Model), whereas the WE executes workflows described in a suitable

process model language, in our case BPMN. The process entities have a high-level description in LRM linked to an implementation entity containing the low-level description, i.e. a BPMN file (see Appendix C Process Entities).

The Process Compiler enables the validation of process implementations, the recompilation of implementation files and the creation of new processes. It supports the definition of a process as a combination of other processes (resulting in an *aggregated process*, cf. Appendix C) with the compilation of its implementation file, providing flexibility in the design of new preservation processes. This combination of processes is carried out by the PC after the user has defined, at a high level, the process structure and flow, such that the specialist expert can concentrate on the creation of useful processes without, in most cases, needing to deal with low-level or implementation detail.

4.2.7 Test Management And Configuration

The Test Management and Configuration Component is an external component used with the integration framework to create a test bed implementation. This component is responsible for configuring the test environment in which the integration framework will be instantiated, and running test suites.

This component reads configuration and test settings, instantiates an environment based upon those configurations, executes the test suites and reports on the results of those tests. This is a common set of requirements for continuous integration and test management software suites available to software developers.

4.2.8 Client applications

The current framework recognises a need for client applications within the PERICLES architecture that provide mechanisms for interacting with and managing the PERICLES models and processes. These client applications include the Policy Editor for editing policies within the managed data and the Model Impact Change Explorer (MICE) to visualise and evaluate the impact upon the controlled data and system of any changes that they intend to perform.

4.2.8.1 POLICY EDITOR

In PERICLES the definition of policy can be stated as:

> *A policy is a high and intermediate level natural language description of an institution's aims/goals that contain constraints on how to achieve the aims/goals.*

Policies are used to ensure that the state of the overall system and objects under control are consistent with the processes and standards required for the running of an archive by the organisation(s) responsible.

The described policy change scenario (Appendix A) delves into the linking of a policy to an actionable process for ensuring the application of a policy. Before getting to the stage of enacting a policy, the policy must first be created by the responsible parties and then added to the archive in some format that is compatible. This is the purpose of the Policy Editor, it will allow a user to create and update policies.

The Policy Editor can be developed to provide templates and guidance on the creation of system compatible policies from high-level abstractions, removing some of the possibility of the introduction of errors by having non-compatible editors.

The Policy Editor will facilitate the derivation of "concrete", detailed policies out of high-level abstract ones. It is being developed as a Java-based web application with the UI based on GWT (Google Web Toolkit).

Deployment of the Policy Editor requires a servlet container (compatible with the Servlet 2.5 API specification) in a Java runtime environment. For facilitating and standardizing deployment, Docker images will be created.

4.2.8.2 MICE (Model Impact Change Explorer)

Model-driven preservation planning is a key part of the PERICLES approach to long-term data preservation, though how changes to processes, policies or formats will impact an existing archive can be hard to evaluate. To assist in this, a client application called the Model Impact Change Explorer (MICE) is under development to assist an archive manager in evaluating and understanding how a potential change to an element in the archive will impact the overall archive. A primary example of where MICE could be used is provided in Appendix A, where this scenario is concerned with the changing of a policy that governs some aspect of the archival system.

MICE will primarily interact with the ERMR, which provides a mediator mechanism to facilitate the reasoning capabilities of the LRM Service for use in evaluating entities and relationships – MICE will then provide a visualisation mechanism to display the type and range of projected impacts upon the entities and digital objects affected by proposed changes.

MICE should be able to communicate with the Workflow Engine and Process Compiler components in order to execute proposed changes. This will allow access to more in-depth analysis through the use of test workflows for proposed changes – this will be done in a sandbox execution environment where changes are not being carried out on the live active data versions but upon copied safe versions. These workflows will be able to supplement the analysis from the ERMR/LRMS with run-time data for approval and evaluation by the archive expert.

4.3 Implementation Status

This section details the current status of the test bed with integration framework. As of September 2015, the following elements are present in the test bed:

- Jenkins
- Docker
- Python-based Web Service Wrapper
- Component Images for Scenario Use
- jBPM Standalone Instance
- Embedded jBPM Engine in Scenario Tests
- Test Cases for Noise Reduction, Video Metadata Extraction

The list is the major sections of work currently being carried out on the integration framework on the test bed. Each will be further described in this section, as to current status and highlight on-going work and maintenance issues.

The ERMR is being developed under aegis of WP5 and is reported upon in Deliverable 5.2.

4.3.1 Test management and configuration: Jenkins & JUnit

4.3.1.1 DEPLOYED TECHNOLOGY

Jenkins is an open source server-based application for continuous integration. It is mainly used to automate software builds, run unattended tests or control other repetitive tasks. Configured tasks are executed regularly in a time-scheduled ("cron-like") fashion or triggered by internal or external events like source code repository changes or dependencies between individual tasks. Build artefacts, reports and statistics are presented in a browser-based web interface and are programmatically accessible via web services. Plug-ins provide additional functionality or integration with other software tools, for example Docker virtualisation or the Maven build system. While it is possible to use Jenkins with other programming languages, the main target is Java.

Java is a general purpose programming language designed to remove as many platform dependencies as possible, so that code can be written for any platform without modification. This is accomplished via the use of Java Virtual Machine and an intermediate byte-code format which Java code is compiled to. Java is object-orientated via classes and interfaces and supports concurrent operations. It supports a wide range of libraries and supporting tools and documentation are available. The Java language was used in the development of jBPM by Redhat thus, with the testing support available, is a reasonable choice to base the test system for the test bed implementation of the integration framework on.

jUnit is a testing framework developed for Java which supports in version 4 annotation based mark-up for Java test code. This allows tests to be coded as standard Java classes and methods and then annotated with tags including '@Test' to indicate a test case, '@Before' and '@After' to indicate functions that need to be carried out before each test.

jUnit has evolved over its development and its reporting and support for testing has been integrated into many different tools such as Jenkins and the Eclipse IDE. It belongs to the xUnit family of testing frameworks which all support similar operations across different languages.

4.3.1.2 STATUS

The PERICLES test bed infrastructure currently provides a fully configured Jenkins server capable of building and running Docker images, automated software builds for all PERICLES specific Maven packages and several integration-tests of the test bed infrastructure itself. Most tasks are triggered by changes in source control and report directly to the affected developers. The Jenkins dashboard provides a comprehensive view on the overall state of the infrastructure and allows to quickly identify failing or unstable tasks.

Currently, only a single Jenkins build server is required and the server setup only includes basic tools and functionality. Tasks with more complex runtime dependencies are run within isolated Docker containers. The base images for these containers are again built and kept up-to-date by Jenkins. Additional build servers may be added later. The move from fully configured build serves to Docker-encapsulated build environments ensures a relatively low maintenance overhead for the infrastructure itself, while enabling arbitrary complex test scenarios with a large number of Docker-managed components. As a positive side effect, this setup offers a high level of protection from misbehaving components caused by security breaches or software bugs. The host system is neither affected nor compromised and inconsistent runtimes can be built from scratch at any time.

Jenkins has been installed and running on the host https://c102-084.cloud.gwdg.de/jenkins/.

This host is secured and can only be accessed with secured networks as determined by GWDG and its network mapping.

The installation is in active use and is used to manage the build cycle for Docker images and to manage and report on the test scenarios being developed in PERICLES.

4.3.1.3 MAINTENANCE

On-going maintenance for Jenkins includes upgrading the plugins, which are used to interact with underlying software such as Docker, and upgrading the core Jenkins product as new releases and patches are released. It should be noted that a key part of this process is ensuring that the changes to Jenkins do not introduce failures in the test system

To prevent major failure due to an upgrade, the Jenkins host is imaged for backup purposes before all major upgrades. This ensures that if the upgrade fails that the server can be brought back online in short order to prevent any issues with on-going work.

4.3.2 Execution layer: Docker

4.3.2.1 DEPLOYED TECHNOLOGY

Docker is an open source application to manage and run applications in isolated environments. It uses operating-system level software-containers in combination with layered file systems and virtual network devices and is considered a lightweight alternative to full hardware virtualisation.

In PERICLES, Docker is used to build and provision the specific runtime environments needed by the various software components automatically and reproducibly, and to run several isolated instances of these components on the same hardware without the risk of interference.

4.3.2.2 STATUS

Docker has been installed for the creation of application containers for use in the test bed for purposes of hosting applications and enabling testing of the integration framework when undertaking scenario-based testing.

The installation is functioning as required and a local copy of required images is stored in the host machine. The software is used to pull a set of application configurations and Docker files (configuration for automated setup of application containers) from the Git repository, hosting the application configurations, details available at https://projects.gwdg.de/projects/pericles-public?jump=my).

4.3.2.3 MAINTENANCE

On-going maintenance includes monitoring of the base Docker images which are used in creating the appliance containers. The containers are based on the Ubuntu Linux distribution and as changes are made to the base image provided by public Docker repositories these must be introduced to ensure Docker images continue to function as expected.

The other main aspect of Docker maintenance is the updating of the Docker software – this is undertaken by GWDG technical staff who monitor when new releases of the software is made available and undertake a risk evaluation before installing the new version if deemed with in acceptable boundaries of risk. This update process includes determining how the Jenkins plugin for communication and control of Docker will interact with the new version and if any conflicts are introduced.

4.3.3 Workflow Engine: embedded jBPM

4.3.3.1 DEPLOYED TECHNOLOGY

Business Process Model and Notation (BPMN) is a representation for specifying business process models, currently ratified to version 2.0 [15]. BPMN's functionality supports the main requirements of the integration framework and the concepts within PERICLES, including human tasks, web-service tasks, parallel 'lane-based' processing, trigger events and decision gateways.

jBPM is the BPMN workflow engine developed by Redhat and the JBoss community. jBPM takes an XML workflow written in BPMN as its input. Data models used in the process notation are based on Java classes for persistence and interaction with data sources. Basic jBPM provides the basic BPMN tasks and support custom task definition.

jBPM supports jUnit testing through helper classes and an embedded runtime engine which allows for fast and simplified BPMN process testing and instantiation with a reduced resource overhead.

4.3.3.2 STATUS

The embedded jBPM engine is provided as part of the jBPM dependencies for use in testing and development. This engine is in use currently with the test scenarios being managed by Jenkins.

The project has deployed a stand-alone jBPM engine. This is used for introduction to jBPM and has full range of the capabilities provided by jBPM including the editor functions. This installation will be used in latter stages of the project to try out high user interaction scenarios on an evaluation basis as opposed to the more regimented continuous integration testing due to the issues of artefact generation and test case influence bleed.

4.3.3.3 MAINTENANCE

This software is monitored by WP6 technical staff who ensure that the software is kept working – a working backup image of the installation has been generated and stored.

Before the end of the project, it is intended that the current jBPM installation will be replaced by the latest version which introduces some changes to its operation and moves from the JBoss Application Server version 7 to JBoss Wildfly.

The maintenance for the embedded engine is naturally linked to the stand-alone jBPM installation – these two components need to be kept in step as using the wrong embedded engine could cause the tested scenarios to fail on the standalone installation.

4.3.4 Handlers: Python-based web service wrapper

4.3.4.1 DEPLOYED TECHNOLOGY

The Python Web Service Handlers are underpinned by the following technologies:

- Python
- Git
- HTTP
- REST

Python is a general purpose, high level programming language, which places emphasis on code readability to allow developers to produce and maintain code across all scales of projects. Python is an interpreted language with a stored compilation format after initial run, the language is

dynamically typed and supports many different programming styles. There is extensive support in third-party libraries for additional functionality beyond the core Python library.

Git is a distributed version control management system that puts emphasis on lightweight controls, efficient fork, branch and repository management mechanisms. Git's distributed nature means that each developer's repository is a full and valid repository containing source code with the ability to pull and push changes from and to other repositories in the same project. This flexibility and lack of a central control mechanism means that patches and fixes can be easily transferred for activities such as testing before being transferred to a full release.

HTTP (Hypertext Transfer Protocol) is a commonly used protocol on the World Wide Web – it is used to transfer data or hypertext (structured text with logical linking nodes) between hosts and clients. This sits in the application layer of the Internet protocol stack above TCP beside FTP and SSH.

REST (Representational State Transfer) is an architecture style for creating resource-based interfaces to systems exposed most commonly through HTTP. It is notably in that this is a client-server architecture with no notion of the client state being stored on the server and that all queries and operations must give the client context to the server to operate with.

4.3.4.2 STATUS

The Python Web Service wrappers developed by DOTSOFT are installed via Docker containers that host the underlying applications. The wrappers have been used and tested in the application-hosting environment and in current use with the testing scenarios.

4.3.4.3 MAINTENANCE

DOTSOFT are responsible for the on-going changes to the python code and maintenance of the wrapper API and documentation. When changes are made to the wrappers, the corresponding Git repository <ref to be finalised> is updated and the application Docker containers can be updated with the new software.

4.3.5 Data storage components

4.3.5.1 DEPLOYED TECHNOLOGIES

Over the course of the last year the project has identified three technologies of particular interest for persistent long-term storage of digital objects within the PERICLES test beds. These three are currently deployed as "special" persistent components in Docker images; they are:

- IRODS (File storage and management);
- CDSTAR (Management and Files);
- PostgreSQL (Database).

Status and maintenance are handled in the same way as with other components (see below).

4.3.6 Component images for scenario use

4.3.6.1 STATUS

This is an on-going task – the component images which are applications in Docker containers are created with Docker file with associated configuration files and application specific information.

At the time of this deliverable, the following third-party applications and project-developed components are in Docker containers for use in testing scenarios with integration framework compatible communications:

- General Applications
  - BagIt (File Packager)
  - Exif Tool (Image Information)
  - FIDO (File identification)
  - MediaInfo (Video File Metadata)
  - ClamAV (Antivirus)
  - FFMpeg (Video Tool)
  - Git (Repository software)
  - Java (Basic Java Language support)
- Project Applications and Components
  - Noise Reduction (Scenario Software (section B))
  - MediaInfo Report (Scenario Software to produce readable and usable output)
  - Python Web Service Wrapper (Common Component for communications)
  - BagIt File Store (Mockup storage layer)

These applications have been configured for use in the project via the requirements gathered from WP2-5.

4.3.6.2 MAINTENANCE

The maintenance of the Dockerised applications is a responsibility of WP6 technical staff who are required to ensure that the Docker images build and function correctly. Jenkins is used to automate the build process and the containers are tested through use in the scenarios. When an issue occurs, the initial point of contact for change is the Docker image creator, though in failing that another member of WP6 will be assigned to fix the issue if possible.

4.3.7 Test cases

4.3.7.1 STATUS

The two main test cases under examination currently are Noise Reduction from the Space Science use domain (see Appendix B) and the Video Metadata Extraction and Consistent Playback scenario (based around work carried out by Dave Rice on behalf of Tate [14]) – both of which are centred around an automated structured approach to populating the metadata and models for the entities within a PERICLES system. The latter stages of these scenarios will deal with the updating of the objects under control due to subsequent operations.

These two cases are currently being implementing and controlled by the Jenkins installation with technical staff from WP6 and use case experts and technical staff from WP2-5.

4.3.7.2 MAINTENANCE

These test cases are under active development – any maintenance for them will fall under the banner of continual development.

4.3.7.3 ADDITIONAL CASES

Future additional test cases will be added to the test system as the scenarios and technologies become more mature in terms of the project timeline. These cases will be managed in joint effort by

the scenario originators (WP2-5) and system integrators (WP6). Each test case will need to provide the information asked for in Section 3.2.1, and WP6 will help with getting scenarios operational on the test bed.

5 Communication APIs

We have taken a protocol and interface-centric approach to the integration framework, and its operation thus centres on the communications that are possible between the major components, processing elements and client applications. The integration framework is designed to unify the communications scheme so that each component has a clear communications protocol with defined operations and payload formats. These APIs are described in subsequent subsections for each of the main components ERMR, the Workflow Engine, the Process Compiler and the Data Storage layer.

All processing elements are wrapped by the Web Service Handlers. The client applications that make calls to the major components are:

- MICE
- Policy Editor

The major components have defined communication APIs which are unique to their function – this is to accommodate the broad range of functionality that is built into these components, particularly in the case of the Workflow Engine where an existing API and implementation are used, it makes no sense to rewrite or mask something that is compatible. This is a similar case with data storage, technologies such as iRods have defined communications formats which can be reused with no major design changes. Only the ERMR and Process Compiler are PERICLES sourced unique APIs in the major components.

The client applications will make broad use of the querying mechanisms exposed by the other components, for example the Policy Editor and MICE can be used to query the ERMR, call the Process Compiler or launch a process on the Workflow Engine.

All processing elements are wrapped by a web service with a RESTful API utilising a JSON payload format to transfer additional data required. This design choice was introduced in Section 5.3 of D6.1; the choice was made to wrap the processing elements to simplify the creation of process workflows by only having one communication method with a clearly defined format for payload. Another reason for the decision was that not all the applications being utilised have remote calling capability so would either have to share a host with the workflow engine (poor resource use) or have a unique interface created. The wrappers simplify this, with the only change needs to run a component being the configuration of the handler.

5.1 Unified Approach For Processing Units

5.1.1 Handler operation

Handlers must perform operations to fetch remote data via repository or HTTP, execute underlying components according to the payload commands and then publish the results to either a HTTP address or repository.

For flexibility, data transfer methods are per payload via the options in the handler configuration. Every payload post data contain a source URL from which the handler fetches the data to be processed. When a payload is successfully finished, it is transferred to the destination URL, where subsequent processing elements can find the processed data. The source and destination methods can be different, i.e. one can be a repository and the other an HTTP address.

The handler status and processing errors codes can be found in Table 8 and Table 7 respectively in Appendix D.

5.1.2 Configuration

Handlers are configured through a file-based properties file which is written using YAML. The range of configuration options is located in Table 7 in Appendix D. The configuration file must give the handler its basic properties including its hostname, port and log file location.

This configuration file defines the commands available via the web service front-end, in the example in Section 5.1.2.1 with the available command “convert” taking two mandatory parameters and two optional parameters.

The configuration defines where the local file storage is to be located and how to deal with HTTP and repository transfers.

5.1.2.1 CONFIGURATION EXAMPLE

Figure 5-1 is an example configuration file, which runs a program called ‘convert’ and has the command ‘convert’ with the option for HTTP or repository source and destination.

```
#Handler's configuration: Defines all commands that handler can execute,
acceptable/required params, dirs and urls that are used
---
name: "Handler T21"
version: "0.0.1"
path: "/usr/bin" # Path where component binaries etc are stored.
log_file: /home/tania/logs/handlers.log
log_level: DEBUG
hostname: 127.0.0.1
port: '8008'
before_script: "" # Before every execution, maybe we need to copy the
handler bins or something.
after_script: "" # Before every execution, clean up etc.
commands:
   convert:
       params:
           input:
               under: ""
               required: true
           resize:
               under: "-resize"
           quality:
               under: "-quality"
           output:
               under: ""
               required: true
       flags:
       script: "${CMP_PATH}/convert"
       commit: "Committed msg"
       before_script:
           - "mkdir -p /tmp/${HANDLER_PAYLOAD_ID}"
           - "touch /tmp/${HANDLER_PAYLOAD_ID}/status"
       after_scrit:
           - "echo ${HANDLER_PAYLOAD_ID} finished"
transfers:
   repo:
       local_dir: /tmp
       #dir providing repositories (after processing)
       repos_dir: /home/tania/repos_out/
       #url providing repositories (after processing)
       repos_uri: repo_provider
   http:
       local_dir: /tmp
       files_dir: /home/tania/files_out/
       files_url: http_provider
```

Figure 5-1 Example Configuration

5.1.3 API endpoints

The full technical details of the API endpoints are located in Appendix D. The handlers have three endpoints defined which can be targeted, two of which are GET operations only, the third has a POST and GET operation defined upon it.

The three endpoints, relative to the address http://<server>:<port>, are:

- Handler [/] Handler API root endpoint [GET]
- Payloads [/payloads] Handler Payload entry endpoint [GET/POST]
- Payload [/payload/{id}] Specific Handler Payload entry endpoint [GET]

5.1.3.1 [/] ENDPOINT

The [/] Endpoint will give information on the handler status, whether is it working or in an error state via a GET operation. Handler status codes are available in Appendix D.4

The example response for an idle handler would be:

```
Response 200 (application/json)
     Body{
      "status": {
      "code": 1,
      "msg": "idle"
      }
      }
```

Figure 5-2 [/] GET Response

5.1.3.2 [/PAYLOADS] ENDPOINT

The [/payloads] Endpoint allows two operations. There is a GET operation to retrieve all current information on payloads being operated on, waiting to be processed and stored awaiting deletion on the handler system. A POST operation is defined to allow submission of new work payloads to the handler for processing.

The GET operation will return information about the current payloads in the handler system, Figure 5-3 Sample Payload Listing shows an example list of payloads which return the information according the object structure and tags given in the tables in Appendix D.4.

```
Response 200 (application/json)
         Body[
                {
                  "status": "error",
                  "source": {
                    "url": "https://github.com/tania-pets/demo.git",
                    "type": "repo"
                  },
                  "destination": {
                    "url":
        "http://bf5c5:2d92f846de5db3221b27fd739351666f364cd199@127.0.0.1
        :8008/repo_provider/convert/3999bad76f1443ef89a973ab0e9bf5c5",
                    "type": "repo"
                  },
                  "params": {
                    "input": "eu_flag.jpg",
                    "quality": "90",
                    "resize": "100",
                    "output": "tenmt.jpg"
                  },
                  "error": {
                    "msg": "I/O operation on closed file",
                    "code": 30
                  },
                  "flags": [],
                  "cmd": "convert",
                  "id": "3999bad76f1443ef89a973ab0e9bf5c5"
                },
                {
                  "status": "working",
                  "source": {
                    "url": "https://github.com/tania-pets/demo.git",
                    "type": "repo"
                  },
                  "destination": {
                    "type": "repo"
                  },
                  "params": {
                    "input": "eu_flag.jpg",
                    "quality": "90",
                    "resize": "100",
                    "output": "tsnmt.jpg"
                  },
                  "error": "",
                  "flags": [],
                  "cmd": "convert",
                  "id": "e504d58b5d7f4f9889fe25e3379162a2"
                }]
```

Figure 5-3 Sample Payload Listing

The POST operation will take a payload which is structured like a response payload, but lacking the ID field. The handler populates the ID field. The POST operation will create a new payload in the handler with a unique ID, which will be added to the queue for the underlying component to execute upon.

The subsequent example is for using the example 'convert' command to run over a JPEG image and resize it

```
Request (application/json)
      Body

           {
              "cmd": "convert",
              "params": {
                "input": "eu_flag.jpg",
                "resize": "100",
                "quality": "90",
                "output": "test.jpg"
              },
              "flags": [],
              "source": {
                "type": "repo",
                "url": "https://github.com/tania-pets/demo.git"
              },
              "destination": {
                "type": "repo"
              }
            }
```

Figure 5-4 Example POST Request Payload

After the POST is successfully registered the handler will respond with the payload identification tag which can be used for targeted status information.

5.1.3.3 [/PAYLOAD/{ID}] ENDPOINT

The [/payload/{id}] endpoint is a targeted information request for the payload identified by the value {id}. This will return the information about that specific payload to the caller as the result of a GET operation.

The ID will take the form of an alphanumeric string, for example:

```
e504d58b5d7f4f9889fe25e3379162a2
```

The response to the GET operation will take the form of:

```
Response 200 (application/json)
Body
{
   "cmd": "convert",
   "params": {
     "input": "eu_flag.jpg",
     "resize": "100",
     "quality": "90",
     "output": "test.jpg"
   },
   "flags": [],
   "source": {
     "type": "repo",
     "url": "https://github.com/tania-pets/demo.git"
   },
   "destination": {
     "type": "repo"
   }
 }
```

Figure 5-5 Targeted Information Response

5.2 ERMR Communications

5.2.1 Summary

The ERMR is one of the main fixed points in a real world implementation of a PERICLES framework, it acts as an information hub which can be queried for metadata, object locations and through the use of a mediator used as an entry point to the reasoning capabilities of semantic services such as the LRM Service.

The ERMR has to provide the functions to store and update models, store and update entities and their associated metadata, provide query mechanisms to search for entity information.

This component acts as a contact point in the system for anything that needs to obtain the required information for retrieving an object under control in the system – thus it needs to be able to provide the resource locations for any object registered in its locations.

5.2.2 API summary

This summary focuses on the main aspects of the ERMR API that are being exercised in the tests on the test bed. Note that all tables of the request and response headers and values are presented in the associated Appendix for the ERMR.

5.2.2.1 Store New Model or Entity (Creating a new Object)

To create a new model using a file as a source, the following request shall be performed:

PUT <root URI>/api/cdmi/<CollectionName>/<DataObjectName>

- <root URI> is the path to the registry.
- <CollectionName> is zero or more intermediate collections that already exist, with one slash (i.e., "/") between each pair of collection names.
- <DataObjectName> is the name specified for the data object to be created.

5.2.2.1.1 Example

PUT to the collection URI the data object name and contents:

```
PUT /api/cdmi/ModelCollection/TestModel.txt HTTP/1.1
Host: c102-086.cloud.gwdg.de
Accept: application/cdmi-object
Content-Type: application/cdmi-object
X-CDMI-Specification-Version: 1.1
      {
            "mimetype" : "text/plain",
             "metadata" : { ...
                },
            "value" : "This is the Value of this Data Object"
       }
```

```
Response
           HTTP/1.1 201 Created
           Content-Type: application/cdmi-object
           X-CDMI-Specification-Version: 1.1
           {
              "objectType" : "application/cdmi-object",
              "objectID" : "00007ED90010D891022876A8DE0BC0FD",
              "objectName" : "TestModel.txt",
              "parentURI" : "/ModelCollection/",
              "parentID" : "00007E7F00102E230ED82694DAA975D2",
              "mimetype" : "text/plain",
              "metadata" : {
                  "cdmi_size" : "37"
              }
           }
```

5.2.2.2 GET MODEL

The following HTTP GET reads from an existing object at the specified URI:

- GET <root URI>/api/cdmi/<CollectionName>/<DataObjectName>
- GET <root URI>/api/cdmi/<CollectionName>/<DataObjectName> ?value:<range>;
- GET <root URI>/api/cdmi/<CollectionName>/<DataObjectName> ?metadata:<prefix>;

Where:

- <root URI> is the path to the CDMI cloud.
- <CollectionName> is zero or more intermediate collections.
- <DataObjectName> is the name of the data object to be read from.
- <range> is a byte range of the data object value to be returned in the value field.
- <prefix> is a matching prefix that returns all metadata items that start with the prefix value.

5.2.2.2.1 Example

GET to the data object URI to read all fields of the data object:

```
GET /api/cdmi/MyCollection/MyDataObject.txt HTTP/1.1
Host: c102-086.cloud.gwdg.de
Accept: application/cdmi-object
X-CDMI-Specification-Version: 1.1

Response:
           HTTP/1.1 200 OK
           X-CDMI-Specification-Version: 1.1
           Content-Type: application/cdmi-object

           {
              "objectType" : "application/cdmi-object",
              "objectID" : "00007ED90010D891022876A8DE0BC0FD",
              "objectName" : "MyDataObject.txt",
              "parentURI" : "/MyCollection/",
              "parentID" : "00007E7F00102E230ED82694DAA975D2",
              "mimetype" : "text/plain",
              "metadata" : {
                  "cdmi_size" : "37"
              },
              "valuerange" : "0-36",
              "value" : "This is the Value of this Data Object"
           }
```

5.2.2.3 CREATE REPOSITORY IN TRIPLE STORE

To create a new repository, the following request shall be performed:

- PUT <root URI>/api/triple/<NewRepositoryName>

where:

- <root URI> is the path to the registry.
- <NewRepositoryName> is the name for the repository to be created.

5.2.2.3.1 Response Status

HTTP Status	Description
201 Created	The new repository was created

Table 1 Response Status

5.2.2.3.2 Example

PUT to the triple store URI to create a repository:

```
PUT /api/triple/MyRepository HTTP/1.1
Host: c102-086.cloud.gwdg.de

Response:
HTTP/1.1 201 Created
```

5.2.2.4 ADD TRIPLE TO REPOSITORY

The following HTTP PUT add triples to a repository:

- PUT <root URI>/api/triple/<repositoryName>/statements

Where:

- <root URI> is the path to the registry.
- <repositoryName> is the name of the repository.

5.2.2.4.1 Request Headers

Header	Type	Description	Requirement
Content-Type	Header String	The content type of the triples • "text/plain" for ntriples	Mandatory

		• "application/rdf+xml" for RDF	

Table 2 Request Headers

5.2.2.4.2 Request Body

The request message body contains the data to be stored.

5.2.2.4.3 Response Status

HTTP Status	Description
201 Created	The triples were added

Table 3 Response Status

5.2.2.4.4 Example

PUT triples to the repository URI:

```
PUT /api/triple/MyRepository/statements HTTP/1.1
Host: c102-086.cloud.gwdg.de
Content-Type: text/plain

<http://www.pericles.org/models#process1>
<http://www.pericles.org/models#name> "Ingest" .
<http://www.pericles.org/models#process1>
<http://www.pericles.org/models#description> "Ingest documents in
the registry" .
<http://www.pericles.org/models#process1>
<http://www.pericles.org/models#identity> "7d4e14c8-1adf-4cfd-b0b8-
ede46944b006" .
<http://www.pericles.org/models#process1>
<http://www.pericles.org/models#version> "0.1" .

Response:
HTTP/1.1 201 Created
```

5.2.2.5 *List Triples in a Repository*

To list existing repositories, the following request shall be performed:

- GET <root URI>/api/triple

where:

- <root URI> is the path to the registry.

5.2.2.5.1 Response Message Body

The response message contains a JSON list of repository information (to be refined).

5.2.2.5.2 Response Status

HTTP Status	Description
200 OK	The list is returned

Table 4 Response Status

5.2.2.5.3 Example

GET a list of repositories:

```
GET /api/triple HTTP/1.1
Host: c102-086.cloud.gwdg.de

Response:
HTTP/1.1 200 Ok

[ { "title": "DemoLondon", "writable": true, "id": "DemoLondon"
},
 { "title": "Test", "writable": true, "id": "Test" } ]
```

5.2.2.6 Query a Repository (using SPARQL)

To send a SPARQL query to a repository, the following request shall be performed:

- GET <root URI>/api/triple/<repositoryName>?query=<SparqlQuery>

where:

- <root URI> is the path to the registry.
- <repositoryName> is the name of the repository to query.
- <SparqlQuery> is a sparql query encoded as a URI.

5.2.2.6.1 Response Message Body

The response message contains a JSON dictionary:

- “values stores a list of tuples
- “name” stores a list of names for the returned tuples

5.2.2.6.2 Response Status

HTTP Status	Description
200 OK	The list is returned

Table 5 Response Status

5.2.2.6.3 Example

GET to evaluate a SPARQL query on a repository:

```
GET
/api/triple/MyRepo?query=select%20?s%20?p%20?o%20%7B?s%20?p%20?o%7
D HTTP/1.1
Host: c102-086.cloud.gwdg.de

Response:
HTTP/1.1 200 Ok

{
      "values": [
      [
            "<http://www.pericles.org/models#version>",
            "\"1.0\""
      ],
      [
            "<http://www.pericles.org/models#identity>",
            "\"3ecdb028-40ec-453a-b4eb-21ad9234ac5e\""
      ],
      [
            "<http://www.pericles.org/models#name>",
            "\"Convert\""
      ],
...
      ],
      "names": [
      "p",
      "o"
      ]
}
```

5.2.3 Implementation summary

The ERMR is being developed as part of WP5, T5.1 and T5.2. The implementation status and design are reported upon in Deliverable D5.2. There is a demonstration instance of the ERMR available at https://c102-086.cloud.gwdg.de.

The ERMR implements the Cloud Data Management Interface (CDMI) standard and supports SPARQL and triple store operations via a RESTful API.

The next major element of the ERMR will be the mediator functionality to allow the LRM Service to be contacted via the ERMR interface by other components. This will facilitate the loading and interrogation of models and entities via LRM without having to expose the full LRM Service to the wider PERICLES architecture.

5.3 Workflow Engine Communications

5.3.1 Motivation

The workflow engine in the test bed implementation will be provided by jBPM. The main purpose of the workflow engine is to coordinate the required worker components in an archival system or test bed in order to fulfil the specified process. The workflow engine does not deal with the data directly – it deals with the control flow of the process with the data flow being directed by the workflow engine but the actual data transfers occurring on a lower level that the control flow. As such the workflow engine needs to provide the following operations:

- Execute/Cancel/Retrieve Status Process
- Execute/Cancel/Assign/Claim Retrieve Process Task
- Retrieve Running/Ended Processes
- Retrieve Deployed Processes
- Deploy/Un-deploy Process

The workflow engine becomes one of the fixed points in the design – it is a contact point anything that needs to get work executed needs to know about and in the framework this puts it solidly as a coordinator of other services.

5.3.2 API summary

A full jBPM REST API summary can be found on the jBPM documentation website [3]. In this section we highlight key parts of the API which help with the tasks of executing a process, retrieving currently deployed processes, claim a process task and listing all process instances.

The REST API is being used as it offers the functions that PERICLES requires and the server technology can be disconnected from the client technology – meaning that if either one changes behind the interface – the operation is preserved.

Note: All classes referred to are jBPM implementation classes.

All the calls are made relative to the URL http://server:port/jbpm-console/rest where *server* is the jBPM host and *port* is the port on which the server is available.

5.3.2.1 Retrieve Currently Deployed Process Groups

This operation will list the currently deployed process groups on the jBPM server. This will be grouped into different process groups as determined by the system administrators and technicians.

For further information on a specific deployment – and additional parameter can be added which is the deployment identifier.

[GET] /deployment/

- Returns a list of all the available deployed processes as an XML document, eg.

```
<deployment-unit-list>
      <deployment-unit>
            <groupId>eu.pericles.envmon</groupId>
            <artifactId>EnvironmentMonitoring</artifactId>
            <version>1.0</version>
            <kbaseName/>
            <ksessionName/>
            <strategy>SINGLETON</strategy>
            <status>DEPLOYED</status>
      </deployment-unit>
</deployment-unit-list>
```

[GET] /deployment/{deploymentId}

- Returns specified deployment for deploymentId as an XML document, eg.

```
<deployment-unit>
      <groupId>eu.pericles.initialcontact</groupId>
      <artifactId>InitialContact</artifactId>
      <version>1.0</version>
      <kbaseName/>
      <ksessionName/>
      <strategy>SINGLETON</strategy>
      <status>DEPLOYED</status>
</deployment-unit>
```

5.3.2.2 LIST ALL PROCESS INSTANCES

This operation should list all current and recently completed process instances in the jBPM server. This operation will continue to should an instance after it completes until the process instance cache limit is reached in which case completed processes will be removed or until a user deletes a process explicitly.

[GET] /history/instances

- Gets a list of process instances that appear on the jBPM host. (ProcessInstanceLog)
- Can be paginated, eg.

```
<log-instance-list>
   <process-instance-log id="1">
         <process-instance-id>1</process-instance-id>
         <process-
         id>EnvironmentMonitoring.PlaybackMonitorIngest</process-id>
         <start>2015-01-20T09:27:21+01:00</start>
```

```
            <end>2015-01-20T09:27:21+01:00</end>
            <status>2</status>
            <parent-process-instance-
            id xmlns:xsi="http://www.w3.org/2001/XMLSchema-
            instance" xsi:nil="true"/>
            <outcome xmlns:xsi="http://www.w3.org/2001/XMLSchema-
            instance" xsi:nil="true"/>
            <duration>79</duration>
            <identity>tpetso</identity>
            <process-version>1.0</process-version>
            <process-name>PlaybackMonitorIngest</process-name>
            <external-
            id>eu.pericles.envmon:EnvironmentMonitoring:1.0</external-id>
      </process-instance-log>
   </log-instance-list>
```

For a specific instance, the following would be used:

[GET] /history/instance/*{procInstId}*

- Gets the process instance associated with the specified process instance
- *procInstId* must conform to the following regex: [0-9]+

5.3.2.3 EXECUTE PROCESS INSTANCE

To execute a process instance the following target should be used:

[POST] **/runtime/*{deploymentId}*/process/*{processDefId}*/start**

- Starts the process processDefId from the deployed group deploymentId
- Takes input parameters in a map query parameters
- Only basic types can be used – complex types need the /execute REST call (see jBPM documentation)
- Returns a process instance dependent on what process is used.

5.3.2.4 CLAIM PROCESS TASK

For human or approval tasks, individuals either via a web interface or other client program needs to be able to check for and claim tasks from the workflow engine.

To check the list of available tasks the following is used:

[GET] /task/query

- This operation queries the task manager based on a set of given parameters including potential owner, process instance identifier and status. A full listing can be found in the jBPM documentation.
- This returns a task summary list.
- This list contains tasks, which have not been archived within jBPM and are available to view.

To claim a task:

[POST] /task/*{taskId}*/claim

- This will assign the task taskId to the calling application user.
- This will return a status bean which will inform the user if their claim has been successful

Subsequent calls for these tasks will be:

[POST] /task/*{taskId}*/start

- Starts task taskId

[POST] /task/*{taskId}*/stop

- Stops task taskId

[POST] /task/*{taskId}*/complete

- Completes a task
- This operation will take map query parameters which will be used to complete the task as the results of the task. Only basic types can be done this way – complex or custom types require the execute command.

5.3.3 Implementation summary

The jBPM BPMN workflow engine is developed and maintained by the JBoss community. Currently version 6.1.0 is installed as a standalone instance and the version 6.1.0 embedded runtime is used for testing purposes.

Version 6.2.0 and the underlying server software Wildfly will be evaluated for use in the test bed over the next 12 months and if found to have minimal impact should be installed and the test bed updated to this version.

Note that PERICLES did not rewrite the jBPM REST API and is reusing a publically available toolset. The major adaptions will come when the Process Compiler and Workflow Engine are being used to create and update processes within a PERICLES system as at this point the deployment process will differ from the standard jBPM process deployment due to the additional testing and validation processes that PERICLES is introducing.

5.4 Process Compiler Communications

5.4.1 Motivation

When a user needs a new process to perform a specific task, three scenarios are possible:

- The process can be defined by modifying an available process in the ecosystem: the user modifies the process in the ME (Model Editor), which can be recompiled by the Process Compiler to update the executable file containing the low-level description of the process.
- The process can be defined as a combination of other available processes in the ecosystem: using the ME, the user creates the new process as an aggregated process entity by specifying input and output entities, the sequence of sub-processes as well as the input and output data flows needed to perform the operation. This aggregated process is compiled to create an executable file containing the low-level description of the process.
- The process can be described neither as a modification nor as a combination of other available processes in the ecosystem: using the ME, the user creates the new process as an atomic process entity by specifying the input and output entities, the operators used to perform the process and the implementation file associated, which has to be created beforehand, i.e. using the editor provided by jBPM in the case of BPMN implementations. The PC can be used to validate both descriptions.

The Process Compiler is an execution layer component instantiated by the WE (Workflow Engine) when necessary during the execution of a workflow. It is launched by the WE and performs read-only operations over the ERMR (Entity Registry Model Repository).

5.4.2 API summary

The communications with the ERMR will use the standard ERMR communications and query mechanisms as detailed in Section 5.3. At the first stage, the PC will be implemented as a simple command-line tool that can be wrapped by a handler. At more advanced stages an own REST API supporting XML and JSON formats will be provided to speed up the communications.

The operations the WE can request to the PC are:

1. Compile an aggregated process
2. Recompile a process (both aggregated and atomic processes)
3. Validate an implementation (both aggregated and atomic processes)
4. Validate a sequence (only aggregated processes)

5.4.2.1 REQUEST

All these operations are requested via POST. The necessary data to perform the request can be sent in the form of a XML or JSON input file containing all the information:

```
+ Request (application/json)
+ Body
{
   "cmd": "recompile",
   "params": {
      "input": "input_file.xml"
   },
   "source": {
      "type": "repo",
      "url": "https://pericles@git.projects.gwdg.de/model-
compiler.git"
   }
}
```

or specifying the values needed in the body of the request:

```
+ Request (application/json)
+ Body
{
   "cmd": "recompile",
   "params": {
      "configuration_data": {
      "path": https://c102-086.cloud.gwdg.de
      "access_token": "sqbkktg3s00vzz7gg3s198rzb9g3s2me2u2ng3s3"
      },
      "process_id": "02eb9874-4fb0-11e5-885d-feff819cdc9f"
   }
}
```

The first format is more suitable when compiling a process for the first time (*Compile an aggregated process* request) or when changes to process entities have not been populated yet. The second format is more suitable when changes to entities have been already populated, so they all can be referred directly by their identifiers.

5.4.2.1.1 Compile an aggregated process

The data input in the request is:

- Command: “compile”
- Configuration data: path to the location where the resources are stored (i.e. ERMR URL) as well as credentials or access token.
- Aggregated process description (see Appendix C Process Entities for details)

5.4.2.1.2 Recompile a process

The data input in the request is:

- Command: “recompile”
- Configuration data: path to the location where the resources are stored (i.e. ERMR URL) as well as credentials or access token
- Process identifier or description (see Appendix C Process Entities for details)

5.4.2.1.3 Validate an implementation

The data input in the request is:

- Command: “validate_implementation”
- Process information:
  - Configuration data: path to the location where the resources are stored (i.e. ERMR URL) as well as credentials or access token
  - Process identifier or description (see atomic and aggregated process attributes in Section Process Entities)
- Implementation information:
  - Configuration data: path to the location where the resources are stored (i.e. ERMR URL or Data Store URL) as well as credentials or access token
  - Implementation identifier or implementation file (i.e. BPMN file, see Appendix C Process Entities for details)

5.4.2.1.4 Validate a sequence

The data input in the request is:

- Command: “validate_sequence”
- Configuration data: path to the location where the resources are stored (i.e. ERMR URL) as well as credentials or access token
- Aggregated process identifier or description (see Appendix C Process Entities for details)

5.4.2.2 RESPONSE

In case of success, the PC sends a JSON response specifying at the body the appropriate code and the location of the implementation file in the case of a compilation/recompilation request:

```
+ Response 201 Created (application/json)
+ Body
{
   "params": {
      "output": "new_process.bpmn2"
   },
   "destination": {
```

```
        "type": "repo"
        "url": "https://pericles@git.projects.gwdg.de/model-
compiler.git"
    }
}
```

or the validation result in the case of a validation (sentence/implementation) request:

```
+ Response 200 OK (application/json)
+ Body
{
    "valid": "false"
    "reason": "Required input missing"
}
```

5.4.2.3 ERROR

In case of error, the PC sends a JSON response specifying at the body the appropriate error code and the reason of error:

```
+ Error 404 Not Found (application/json)
+ Body
{
    "error": "not_found",
    "reason": "missing"
}
```

5.4.3 Implementation summary

This section describes a work-in-progress, therefore future refinements and changes will be done during the course of the project.

5.4.3.1 COMPILE AN AGGREGATED PROCESS

To create a new executable workflow as a combination of other processes using the Process Compiler the user only has to create an aggregated process by defining its expected inputs and outputs and the sub-process sequence (see Appendix Process Entities). Each step of this sequence is represented by a sub-process and the mapping between its input and outputs with the available resources at that time, that is input resources and resources created in the previous steps. After that, the Process Compiler compiles the aggregated process to create an implementation file by doing the following:

1. Read input file
2. Fetch all referenced sub-processes from the repository.
    a. Validate checksum for source process files.
3. Parse implementation descriptions of sub-processes.
    a. Validate file structure against the implementation specification.
    b. Reject input with unknown/unsupported attributes or tags.
    c. Identify start/end events, error events, input/output data flows and data entity types.
4. Populate a list of available resource URIs with the input resources.
5. For each sub-process in the sequence:

   a. Connect input data flows with Digital Resource URIs according to the input mapping.
      i. Check entity compatibility (otherwise: type-error) => this may imply to check the inheritance tree.
      ii. Check that all used resources are available at that point (otherwise: resource-not-available-error).
   b. Connect output data flows with temporary Digital Resource URIs according to the output mapping.
      i. Check entity compatibility (otherwise: type-error) => this may imply to check the inheritance tree.
      ii. Check that these URIs are not present in the list of available resources (otherwise: resource-overwrite-error).
   c. Update the list of available resources with the new resources.
6. Create a new implementation file for the aggregated process.
   a. Control flow handling.
      i. New start-event points to start-event of first sub-process.
      ii. End-event of last sub-process points to new end-event.
      iii. Populate and connect sub-processes descriptions.
      iv. Error handling.
   b. Data flow handling.
      i. Add the resource data flow definitions identified previously.
      ii. Data casting.
7. Store file in a local storage.
8. Return new implementation file location to the WE.

5.4.3.2 RECOMPILE A PROCESS

If the process is an aggregated process, the recompilation is done is in the same way as the compilation (see above). If the process is an atomic process, the Process Compiler does the following:

1. Read process entity: by reading the input file or by accessing to the process entity and searching for its values in the ERMR (depending on the request format).
2. Read implementation file: if it was not specified in the request body, search for it in the ERMR.
   a. Validate file structure against implementation specification.
   b. Reject input with unknown/unsupported attributes or tags.
   c. Identify process elements of interest: data input and output, data objects and operators.
3. Validate inputs and outputs: check entity compatibility and loose ends.
   a. Convert data types when necessary.
4. Validate operators: check compatibility between input/output/data objects and data types supported by operators.
   a. Convert data types when necessary.
   b. Modify communication interfaces when necessary.
5. Write a new implementation file populating these changes.
6. Store file in a local storage.
7. Return new implementation file location to the WE.

5.4.3.3 VALIDATE AN IMPLEMENTATION

To validate a process entity and its implementation, the Process Compiler does the following:

1. Read process entity: by reading the input file or by accessing to the process entity and searching for its values in the ERMR (depending on the request format).
2. Read implementation file: if it was not specified in the request body, search for it in the ERMR.
    a. Validate file structure against implementation specification.
    b. Reject input with unknown/unsupported attributes or tags.
    c. Identify process elements of interest: data input and output, data objects and operators.
3. Validate inputs and outputs: check entity compatibility and loose ends.
    a. If aggregated process, validate sequence (see below).
4. Validate operators: check compatibility between input/output/data objects and data types supported by operators.
    a. For aggregated process, if the sequence contains aggregated sub-processes decompose them until all the sub-processes are atomic to create the list of operators.
5. Return validation result.

5.4.3.4 VALIDATE A SEQUENCE

To validate the sequence of an aggregated process entity, the Process Compiler does the following:

1. Read process entity: by reading the input file or by accessing to the process entity and searching for its values in the ERMR (depending on the request format).
2. Fetch all referenced sub-processes from the repository.
3. Populate a temporal list of available resource URIs with the input resources.
4. For each sub-process in the sequence:
    a. Connect input data flows with Digital Resource URIs according to the input mapping.
        i. Check entity compatibility.
        ii. Check that all used resources are available at that point.
    b. Connect output data flows with temporary Digital Resource URIs according to the output mapping.
        i. Check entity compatibility.
        ii. Check that these URIs are not present in the list of available resources.
        iii. Update the list of available resources with the new resources.
5. Return validation result.

5.5 Client Applications

5.5.1 Policy Editor

5.5.1.1 MOTIVATION

The Policy Editor is being developed using an iterative design process guided by the continuing requirements developed in WP2-5. With the Policy Editor being a user-facing component in the framework as a client application, the need for clearly defined communication channels between it and other components is essential – this is due to the nature of the component needing to be flexible in design but having clear behaviours to deal with and create. The Policy Editor is being designed to work both as a stand-alone application and as an integrated component for a PERICLES-based system with an ERMR and external data stores.

Integration will happen by means of third-party adapters, which the Policy Editor will define clear APIs to adhere to meaning that the editor can be used not only within a PERICLES system but with other systems which create an adapter which is compatible.

An interaction set with other component types have been identified and form the basis of the proposed design summary. Each adapter will have to implement operations to interface with data managed by the respective component data and entity stores necessitating in-built format conversions where required.

5.5.1.2 Design Requirement Summary

5.5.1.2.1 Data Types

The Policy Editor needs to support the following entity types:

- Policy
- Process
- Template

Each type will be defined in the Ecosystem and Domain Specific ontologies as defined by WP2/4. The Policy Editor needs to be able to access the definitions of these entities through an adapter.

5.5.1.2.2 Component Interactions

The main components the Policy Editor will require to communicate with are the Workflow Engine, ERMR and Data Storage Components.

Workflow Engine

The Policy Editor needs to be able to launch workflows via the Workflow Engine for the following purposes:

- Store New Policy
- Update Policy
- Retire Policy
- Attach Existing Process
- Compile Policy Process

ERMR

The Policy Editor needs to be able to access the ERMR for the following reasons:

- Search for Policy
- Get Related Policy Information
- Access Version Information

Data Storage

The Policy Editor needs to access data storage to retrieve any additional files which are required during the policy editing and creation stages – these could be archived versions, supporting documentation or templates.

5.5.1.2.3 Adapter Design Summary

While the Policy Editor can be integrated into the PERICLES framework it does this via an adapter which allows the editor to, in theory, be used in any policy based system – which PERICLES can fall into as policies form part of the model-driven approach being undertaken.

Adapters will have to implement the following operations (note these may be subject to change):

Operation	Description
loadItem(identifier)	Loads an Item from the target component using the given identifier value
addItem(identifier, data)	Stores an item under the given identifier with the values in data
updateToVersion(identifier, version, data)	Updates an existing item indicated by identifier value to the new version and data
updateProperties(identifier,properties)	Updates the properties of the item with the given identifier with the property data values in properties
searchByProperty(property,value)	Searchs the target store for items that match the given value for the given property.
storeDraft(identifier,data)	Adds a new draft version of a policy to the store but does not make it active.
makeDraftActive(identifier)	Makes any stored draft version of policy active
loadPolicyGraph(identifier)	Load all referenced policies and templates for a given identifier.

Table 6 Adapter Methods

5.5.2 MICE

5.5.2.1 MOTIVATION

As stated in previous sections, the Model Impact Change Explorer is being developed to assist archive managers in evaluating and understanding how potential changes to elements in an archive could have an impact on the overall archive.

Understanding how changes to policies, processes or even individual objects under control could affect an archive is something which is not readily understood, though through the model-driven approach adopted here, it is intended that using a tool such as MICE, archive managers can estimate the impact of any operations they wish to enact before having to carry them out

To this end, MICE is a visualisation and querying mechanism which will interact with the various components of the PERICLES framework in order to present the user with the information required to make informed decisions about how changes will affect the status of the archive.

MICE is a user facing component which could be considered as a visual front-end for a user to interact with the ERMR and subsequently the LRM Services due to the nature of its function.

5.5.2.2 Operation Summary

The key point about MICE is that it is an origin point for actions as determined by a user. It will call other components and await responses to those calls as opposed to other components making call to it. This is due to its nature as a user interactive tool.

The main interactions that MICE will be required to support are call/response operations to the Workflow Engine and the ERMR. In Figure 5-6, the sequence diagram shows how a user can initiate the analysis and implementation of a Policy change (similar to that described in Appendix A) and the flow of events and control between MICE, ERMR and the Workflow Engine. This diagram shows how the LRM Service is in communicated with via the ERMR as part of an on-going process.

5.5.2.2.1 MICE to ERMR

MICE will required access to the operations supported by the ERMR for:

- Load Entity/Entities – this allows MICE to present a visualisation of the entities and models hosted by the ERMR to the user for consideration
- Query Repository for Graphs – this allows a network of entities to be searched and queried for entities by the ERMR
- Query ERMR for Reasoning Operations – this is to query the ERMR for semantic reasoning operations for determining impact levels

5.5.2.2.2 MICE to Workflow Engine

MICE will require the ability to execute workflows, check and execute tasks on the Workflow Engine in order to support the starting of impact analysis workflows. This will require functions to call the following operations on the Workflow Engine:

- Execute Workflow
- Claim Task
- Complete Task
- Check Task Status
- Check Workflow Status
- Obtain Workflow Result

5.5.2.2.3 Example Operation

The example operation (Figure 5-6) shows a scenario where a user wants to evaluate and then confirm the implementation of a policy change regarding video formats.

The operation begins with the proposed change being loaded from the ERMR as an entity graph (based upon the policy and process models). This will require MICE to query ERMR to obtain this information.

The information is presented to the user, who will then launch a Change Impact workflow which evaluates using the ERMR and the LRM Service the potential impact that the change will have on the archive as known about by the semantic and registry services.

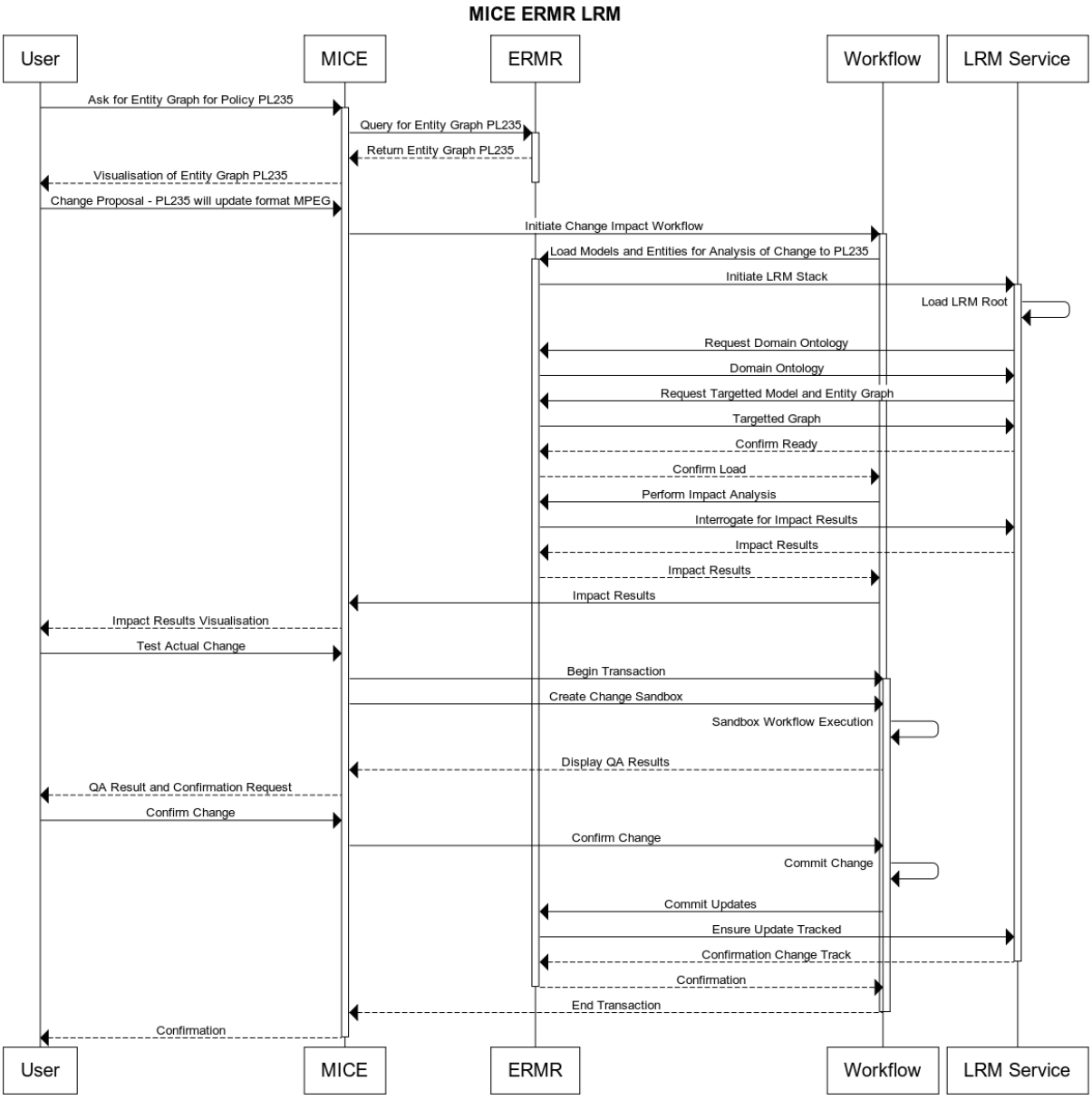

Figure 5-6 Interactions of Client and Major Components

The impact analysis workflow will trigger a number of communications between the ERMR and LRM Service as the LRM Service will need to be loaded with the policy and entity information in order to reason about potential impacts. This will result in new data for a visualisation in MICE to be created allowing the user to either confirm or reject the on-going process.

After the impact has been evaluated MICE could be used to launch a test workflow which will run a sandbox version of any implemented process – this will allow the actual change to be evaluated by the user and system. On the results of this sandbox evaluation, the user can decide whether or not to commit the overall change to the archive.

This is how the ERMR, Workflow Engine, LRMS and MICE could work together in order to accomplish a task.

6 From Test Bed to Real World

6.1 The Differences between Test and Real World

Before moving into discussing how a PERICLES system could be implemented in a real world situation it is important to understand the main differences between the test bed and a real world implementation. This could be a significant number of factors, but this document will concentrate on three main differences:

- "Pristine" Installation
- Controlled Usage Patterns
- Known Fault Induction

These are the main differences between the PERICLES test bed and a real world system that will factor into the discussion for moving from test bed to real world.

PERICLES is not building an archive system – it is developing an approach, which can be adopted by existing systems or used in new systems. Due to this nature of the project, the integration framework on the test bed is not a complete implementation at any one time of the full PERICLES approach. So how does this factor into using the concepts and technologies in PERICLES in a real world system? Can it be integrated or is PERICLES just building some nice demonstrations of no practical use?

6.1.1 "Pristine" installation

One of the major differences between the test bed and a real world system is that for each test run a new 'system' is created. This is to prevent "bleed-over" from test-runs affecting one another – each test is intended to examine a specific area concept, area of functionality or scenario. This means that some scenarios could adversely affect each other – a common occurrence in real world but for the purposes of testing, it is the singular scenario that the project is concerned with.

In the real world, each time you want to run a particular process, the whole system does not get recreated – information has to be retained, other processes will be run at the same time and systems will be shared.

This is a major issue when testing, but it should be noted that it is important to first get the scenario functioning and then integrate multiple scenarios – the multiple scenarios must be held in mind when designing and testing – but the testing of a multiple scenario system is a later stage test rather than the current version of the test bed implementation.

Using a "pristine" system means that any artefacts built up by software during automatic updates, system updates or software failures will be absent – this means that the common side effects of this type of problem will be unlikely to be seen in the testing, as replicating these issues is not a trivial nor possible without real world intervention. This factors into the test system not having the same issues a real world system would have in communicating and operating with an existing system which has procedures and data built up over its operational lifetime.

6.1.2 Controlled usage patterns

The concept of testing introduces the idea of controlled usage patterns – the tests will do a sequence of events in a known order with known data to achieve a known outcome. Controlled usage patterns are useful for proving a set of operations works on given data types and are a repeatable demonstration. Unfortunately, the real world does not operate like this – systems can fail, users can

introduce problems, data types can change and transient errors can occur – all things, which cannot be predicted.

Does organised testing get abandoned? – no, just because not everything can be predicted or matched in behaviour does not mean testing is meaningless. Instead, testing needs to prove that given a reasonable set of factors in the environment the given scenario will operate as expected.

Controlled Usage patterns can be used to show that scenarios will work within a set of parameters – the absence of bugs cannot be proven – only that the known bugs are either solved or present. Similarly for the scenarios – it can be shown that they work for known and 'realistic' conditions but factors can influence this in the real world.

6.1.3 Known fault induction

Testing can detect bugs and errors in code, if those errors are either causing a failure to match expected outcomes or as the result of known faults that will occur in response to different changes or stimuli. Testing can fail due to other factors – unknown errors, transient errors and test system failure – sometimes these will be just unknown problems with the code under test, other times they will have to do with the test environment. This is a problem in the real world errors can originate from outside the actual software system. How does this get dealt with in a test system?

Do we induce errors randomly? We could but how would we check for this? The problem of dealing with errors like this is complicated due to their nature. One solution is to identify the possible failures and induce these at known points and times in the test system. This could include a component failure, data corruption or user error.

Though the induced fault isn't random – it can be used to test how the software and system react to the error. Does the system fully halt, does it handle the error gracefully, can it recover?

In the real world – the errors and faults could occur at anytime – in the test system they occur at known times and the type of error is know as well. This is a major difference in running a test system and a real world system.

6.2 Testing with Confidence

The major differences have been highlighted – so how does the test bed show confidence in the concepts and technologies of PERICLES? The issues and differences are many, so if the test system is different to a real world system how can the results be trusted as an indication of viability?

All software testing outside of actual in-use testing is artificial – it is a set of contrived circumstance to show the viability of software for the task at hand. These circumstances include, but are not limited to, normal, ideal, failure and catastrophic; all of which are to show how the software should handle them.

In PERICLES, the testing is intended to show how the concept of model-driven preservation can be used to meet the requirements from the user domains and how technology can be employed to support this. To this end, the testing in PERICLES can primarily look at three main measures of coverage with regard to testing and the confidence that can be gained:

- Requirements
- User Scenarios
- Design

These three areas are not entirely independent of one another – but can be measured in the three areas.

Requirements coverage – are the requirements created in the project covered by the tests? For each requirement, can a test be linked to it to specify how that requirement is being tested? This may be like 'Can component X read file Y?' where the test is a harness which runs component X against file Y and checks it was read correctly and this can be related to the requirements files of type Y need to be read by the system. If all requirements are linked to tests, which evaluate the system or software under test against those requirements, a level of confidence can be created in the tested product if a) a test exists which examines the requirement and b) the test passes.

User Scenarios – can the software actually fulfil the tasks required by the user? It is important to note that this is subtly different from requirements – as a piece of software can fulfil a list of requirements but not actually be useful for the overall task. This is where user scenarios come in – any acceptance or integration testing in PERICLES should have its roots in user scenarios – these guide how the different components in a system have to work together to achieve a large task. From the testing point of view, this means that the tests for user scenarios have to attempt to fulfil that scenario as a test – does the software do what it is intended to do, does it do it correctly? These are more complex than requirements and the tests become more complex but successful running tests give a higher level of confidence that the software could be used in a production environment.

Design – can the project designs be tested? The design elements in a system based on principles proposed by PERICLES can be complex – often with design elements being introduced to facilitate combinations of requirements or to support maintenance, which could have no direct link to a requirement or scenario. No matter why a design element is present it needs to be tested – how it works, what it is for, is it needed? These tests can be complicated to write – as the structure of the system is being tested as well as the function. Could an element in the design be removed or improved? These are separate from the other two categories as these are dependent on the designers and researchers rather than the users or domain experts. This in itself makes it difficult to do, but if the tests can show that design elements function correctly and fulfil their intended purpose, then confidence in the overall design is improved.

Testing is not infallible – just because all tests pass, does not mean that everything is correct in the written software – only that to the best understanding of the implementers that the produced work can be measured against a set of criteria and found passing. There will be bugs and unknown situations that would be encountered in the real world but testing helps raise levels of confidence in that for most given circumstances the system would work.

6.3 Similarities

This document has highlighted the differences and issues with testing but would be remiss not to discuss the key points of similarities between a test bed implementation and a real world implementation. The key points of similarity are:

- Technology
- Process Patterns
- Data Types

These three key areas in concert with structured and coherent testing allow the demonstration of PERICLES concepts and technology.

6.3.1 Technology

While one of the keys differences was the issue of pristine installation, the fact that the test bed is developed to use the same technology as the real world system is a key motivator in making it able to demonstrate the reliable and stable use of the concepts.

This is an obvious statement, but needs to be clearly stated that building a test bed and testing different technologies to what would actually be used is a pathway to have the work called into question and that the processes would not work.

The caveat to this is sometimes a test system cannot fully replicate the entire technology stack – this could be due to cost, available time and/or space. In these cases the differences in technology need to be explained and to clearly demonstrate how the analogues used will affect performance.

In PERICLES the majority of technologies used are open source and are able to be used in the test bed – where this is not possible, for example with The Museum System (TMS) [8], the collection management system used by the Tate, - a replacement database which mimics the functions of TMS has been included. The project team has determined that this is an acceptable deviation as the database can provide much of the data access required to replace the TMS access.

6.3.2 Process patterns

The processes being developed in PERICLES are processes which are created in concert with the domain experts – the processes are what they would envision the system doing for/with them. This user-focused design methodology underpins the philosophy of how PERICLES is working in WP6 – the test scenarios reflect what the users intend a PERICLES based system would do.

Using the user scenarios as a basis for developing processes for testing the integration framework allows WP6 to test processes as close to real world situations as possible without being in the real world.

This allows WP6 to explore the error pathways in a safe and controlled manner without endangering live data. As the processes are those that would be used in the real world implementation – it helps bolster confidence in the nature of PERICLES concepts and that they could be applied in the real world.

6.3.3 Data types

In addition to the process types being the same, the test data is the same data types and distributions as the real world data. Especially in the case of the Arts and Media domain, getting actual real world data is not possible due to ownership and licensing reasons for the test system, so analogues with the same data types and properties are being used.

While this is synthetic data – it does allow a great degree of testing to be carried out in this project as the data can be generated to match any configuration that the user domain experts can envision – combinations of video containers and codecs, aberrant file formats and other miscellaneous errors which would not be stored in a live system until after cleaning and fixing.

This ability to have the same data types and property distribution while being able to explore rarer or detrimental format types can help with the testing confidence in the process and allows the domain experts to evaluate processes against rare or theoretical cases.

6.4 Evolution of Implementation – not a singular start-up

Although PERICLES concepts could be used to create a new archival system from the ground up for either replacement or starting where no archive existed – it should be noted that the concepts and technology could be used to augment and upgrade existing archives. While superficially these are two very different tasks, the actual processes involved would be quite similar.

6.4.1 Incremental start-up: not everything at once

One of the key implementation concepts PERICLES and WP6 in particular have been examining is that to try and implement a large scale archive system based on the developed principles is a hugely complex task and should be undertaken in an incremental fashion. No trying to bring everything up and online at once – this for many operations would be a key point of failure as there are too many targets and users to account for in what is being developed.

The model-driven approach needs to be used carefully and the models adapted and updated as new entities and concepts enter the archive. This is easier to manage if the archive is developed and built up in smaller segments, dealing with different types of data to be managed in a sequence rather than trying to manage it all at once.

This approach allows implementation and installation errors to be discovered as the system is created rather than having to debug a full installation after it was put in place.

It allows organisations to train users in a systematic fashion which does not place too much of a cognitive load on a user at one time; they only learn about a system as it is made available, not learn about everything at once. This is an easier model for users to learn about things – not trying to absorb concept and action at once for such a complex undertaking.

6.4.2 Use existing systems: bridging

PERICLES can make use of existing systems and indeed does not proscribe any specific technologies for providing underlying functions. This is important as many organisations already have systems for different functions that they are already use and are invested in financially and in knowledge and process, and this can be leveraged. What PERICLES concepts can provide is a way to orchestrate all these technologies in concert to provide greater cooperation and capability across the systems.

This can be found most evident in collection management systems, for example TMS, where a great deal of knowledge in users and about artworks is invested. PERICLES concepts can make use of that knowledge but should not try to replace it outright – over time it may get replaced and PERICLES could help with migration between TMS and another system or version. PERICLES can use it as part of its system to draw upon the knowledge kept there.

In other systems this bridging could extend to third-party services which organisations use on an paid basis – they provide something valuable to the processes being done in that organisation which would be too time consuming or expensive to support internally – the PERICLES model supports this as a core concept, all services can be defined within it and the underlying processes deal with communications.

6.4.3 Map processes: don’t throw out current processes

Too often when new systems are brought in there is a temptation to throw out lots of things that worked – not just in software or hardware – but also in how things are done. Technology and systems should enable experts to do their work more efficiently – not dictate *how* they do that work. When a system is brought in, unless there is a good reason to junk existing processes, the system should support the existing processes.

Will it do everything exactly as the process was before the system? – Probably not, though the method in which a system supports existing processes should enable users to have a recognisable and useable version of existing processes. This mapping of existing process to a PERICLES system is the core of the user scenario testing, taking what the experts do and making the system support that method of working, rather than making the expert change how they work just because the system says so.

This is a key acceptance factor for the users of a system. Ultimately it is upon these users which the success of the new system depends on so if their training and expertise are appropriately leveraged, then the mapping of processes will make successful adoption more probable.

6.4.4 Differences: a chance to improve

While keeping current processes is often a good thing, bringing in a new system or technology should be an opportunity to improve what you are doing. Why does an expert do a process a certain way? Is it just because that is the only way or is it the only viable way up to now? Many experts will work out better and more efficient processes if given the time and technology and training, so when a new system is brought in this is a good opportunity to let experts explore how the processes work.

In the previous subsection, it was discussed about mapping processes to keep existing practices, and for many cases this will work fine, but in changing the system it may be old processes are not possible or become inefficient or too complicated. At this point, think about how it could be carried out in a different, supportable manner that the experts can develop to meet or exceed prior standards.

PERICLES through its model-driven and runtime service instantiation could lead to different approaches to process development – initially many processes will be directly mapped but as the level of knowledge about the approach increases, so too does the possibilities of changes to how things work.

7 Conclusions

The current document has described the final design and current implementation state of the PERICLES integrated test bed. The design goals of the test bed have been to create a framework capable of the flexible execution of varied and varying processing and control components in preservation workflows, while itself being controllable by abstract models of the overall preservation system. To this end if has deployed standard encapsulation technologies – Docker containers and RESTful web services – within a workflow environment – jBPM controlled by Jenkins – and has designed new subsystems to couple these to the abstract models developed in the research activities of the PERICLES project. These new subsystems – the Process Compiler and the Entity Registry Model Repository and its coupling to the LRM Service – provide the means to couple powerful semantic reasoning and policy-driven models to a "live" digital preservation system.

The API designs and technology choices for the test bed are now settled, implementation of the underlying (standard) test bed infrastructure is complete, and implementation of the new ERMR and PC subsystems is well underway. The focus for the integrated test bed over the final stages of the project will be on demonstrating the full end-to-end power of the model-driven preservation approach through the implementation of key application scenarios using models, tools and components drawn from across the PERICLES project.

8 Bibliography

[1] Jenkins – Website Documentation and Description – Tested Active 09/09/2015: https://jenkins-ci.org/
[2] BPMN Standard and Information – Website – Tested Active 09/09/2015: http://www.bpmn.org/
[3] jBPM Documentation and API – Website – Tested Active 09/09/2015: http://www.jbpm.org/
[4] Java API Documentation – Website – Tested Active 09/09/2015: http://docs.oracle.com/javase/7/docs/api/, http://docs.oracle.com/javase/8/docs/api/
[5] JUnit Documentation – Website – Tested Active 09/09/2015: http://junit.org/
[6] Docker Documentation and Software – Website – Tested Active 09/09/2015: https://www.docker.com/
[7] iRODS Documentation and Software – Website – Tested Active 09/09/2015: http://irods.org/
[8] TMS – The Museum System Documentation – Website – Tested Active 09/09/2015: http://www.gallerysystems.com/products-and-services/tms/
[9] PERICLES Initial Version of Testbed Implementation: http://pericles-project.eu/uploads/files/PERICLES_D731_Test-bed_implementation.pdf
[10] Specification of Architecture, Components and Design Characteristics: http://pericles-project.eu/uploads/files/PERICLES_WP6_D61_Specification_of_Architecture_v1.pdf
[11] Python Language Documentation – Website – Tested Active 09/09/2015: https://www.python.org/
[12] Git Version Control Documentation – Website – Tested Active 09/09/2015: https://git-scm.com/
[13] Software Sustainability Institute – Website – Tested Active 09/09/2015: http://www.software.ac.uk/
[14] Sustaining Consistent Video Presentation – Dave Rice, Commissioned by Tate Group http://www.tate.org.uk/research/publications/sustaining-consistent-video-presentation
[15] White, S. A. (2004). Introduction to BPMN. *IBM Cooperation*, *2*(0), 0.
[16] Solem, J. E. (2012). *Programming Computer Vision with Python: Tools and algorithms for analyzing images*. " O'Reilly Media, Inc.".
[17] Weber, G. (1993). USC-SIPI image database: Version 4.
[18] Kaushik, P., & Sharma, Y. (2012). Comparison of different image enhancement techniques based upon PSNR & MSE. *International Journal of Applied Engineering Research*, *7*(11), 2010-2014.
[19] Srivastava, P., Gupta, P., Bhardwaj, S., & Bhateja, V. (2012, September). A new model for performance evaluation of denoising algorithms based on image quality assessment. In *Proceedings of the CUBE International Information Technology Conference* (pp. 5-10). ACM.
[20] Girod, B. (1993) “What’s wrong with mean-squared error,” in Digital Images and Human Vision (A. B. Watson, ed.), pp. 207–220, MIT press.
[21] Wang, Z., Bovik, A. C., Sheikh, H. R., & Simoncelli, E. P. (2004). Image quality assessment: from error visibility to structural similarity. Image Processing, IEEE Transactions on, 13(4), 600-612.

A Appendix – Policy Change Scenario

A.1 Description

We want to change the current policy named Pol-A.v1: "all photos on the website need an ownership annotation (OA)" to a new version Pol-A.v2: "all the photos on the website need an ownership annotation and the Tate logo" to make the provenance of the images visible on the Internet. This scenario contains greater detail than the one described in D5.2.

This is not a stored image production scenario but an access image production scenario. This scenario produces a set of images as the result of adding an OA and the Tate logo to the original images stored in the archive system. The new images are stored in the website, remaining the original ones unchanged.

This type of policy has associated a set of processes {ProAEI, ProEPV, ProA}, where:

- ProAEI is a process that identifies the entities from the ecosystem that have to be validated against the policy
- ProEPV is a process that determines the state of an entity regarding the policy (validated or invalidated state)
- ProA is a process that performs an action over an entity to go from an invalid state to a valid state.

Therefore, we will have defined a set of processes related to Pol-A.v1:

- ProAEI-A: Process "take all the photos published on the website". There is a model on the ERMR that contains the collection of the images, including metadata and a reference to the actual digital objects. There is also a dependency model that describes which of the digital objects have a scaled down copy for the website. The process will use the ERMR to query the relevant photos and pass the result to ProEPV-A.
- ProEPV-A: Process "check if the photo has OA" -> it takes the result set from ProAEI-A as input for the validation.
- ProA-A: Process "add OA to the photo and save the resulting photo on the website" -> it will modify the photo on the web system, which is actually already a copy of the original on a high-value storage.

A.2 Requirements

1. There is no stored validity state. The validated/unvalidated/invalidated state associated to the entities is created (or generated) as the result of querying the ERMR at runtime. The meaning and colours of these states are:
    - validated: the entity has been tested and is now in a conforming state with the ecosystem → green colour
    - unvalidated: the entity has been not yet tested to be valid or invalid → amber colour
    - invalidated: the entity has been tested and is now in a nonconforming state with the ecosystem → red colour
2. Processes operate on the models and entities stored inside the ERMR and on digital objects and bitstreams that are stored in a separate repository/data store.
3. There must be available a set of processes for common operations: update, store, compile, etc.

4. Model editors, such as an RDF Eclipse Plugin, can be used provide information to the user for creation of queries to search entities by value (or set of values) for a specific field.
5. MICE and a model editor could be able to be integrated in the same front end.
6. MICE should allow the user to select entities in the visualisation graph and send them to the a model editor (this being done as MICE exporting the ids of the entities to a file which will be read by the model editor or launching the model editor with arguments). The selected entities are loaded to a model editor for the user to edit, avoiding errors due to manual copy of the entity IDs.
7. After editing the user should be able to launch the WE with inputs the workflow to be executed and a file containing the information changed by the model editor.
8. MICE should allow the user to launch the WE with inputs the entities and the workflow to be executed on them.
9. ERMR must allow transactions; provisional operations that until committed have a clear and achievable undo. During transactions the entities are in the staging area, after approval they are stored in the permanent area.
10. Updates apply to the latest version of an entity unless otherwise specified by a user, who should have the ability to select a different version if needed. Therefore, MICE will display as unvalidated the entities and dependencies related to the last version of the entity.
11. MICE displays a graph related to a specific search, i.e. a specific entity A. Therefore, the state of the dependencies and entities (validated, unvalidated, invalidated) is in regard to the entity A, and when validating an entity the validation is also regarding to its relationship with entity A. MICE allows the user to validate an entity. This validation runs automatically a workflow which updates the dependency between both entities.

A.3 Main Steps

1. Using the Policy Editor, a user makes changes to policy Pol-A: Pol-A.v1 → Pol-A.v2
2. Using MICE, the user checks how this change could affect related entities. MICE will display a visualisation of the policy and its related entities to the user, namely ProAEI-A, ProEPV-A and ProA-A. The entities are marked as unvalidated (amber state) and awaiting user action. The user decides the following:
    a. ProAEI-A is not affected as Pro(take all the photos published on the website) is still the correct process → The user validates ProAEI-A (green state).
    b. ProEPV-A has to be changed to Pro(check if the photo has OA and Tate logo) → we are not going to describe the methodology to change it in this document as we are focusing in the next step, let's assume that a new version ProEPV-A.v2 is successfully created and linked to Pol-A.
    c. ProA-A has to be changed to Pro(add OA and Tate logo to the photo and save the resulting photo on the website) → let's consider this case from now on.
3. Using a model editor, the user asks for available processes that may be useful to change ProA-A by querying via key words for a specific field (or a set of them). For each search, the user queries the ERMR and returns a list of processes that meet the specified conditions.
4. The user wants to create a new process which will add an OA and a logo to an image. He selects the available processes he wants to combine into the new process which are launched as the sequence of the aggregated process and finalises the description of the new process with the required information.
5. The new process entity has to be compiled to generate its BPMN implementation file and store in the ERMR. Therefore, the user launches the WE after editing with a model editor with a workflow which creates, compiles, validates and stores the aggregated process entity and its related implementation entity:

a. Begin Sandbox
b. Start transaction for storing new entities (process entity, implementation entity and dependency entities) in the ERMR
c. Create a process entity, an implementation entity and dependency entities
    i. Populate data to the implementation entity
    ii. Populate data to the process entity from the file containing the process entity description
    iii. Add to the process entity the link to the implementation entity
d. Call the PC with input file containing the aggregated process entity description and compile the process (see Section 5.5.3.1)
e. Populate the BPMN file location to the implementation entity
f. Populate data to dependency entities:
    i. From process entity to implementation entity dependency
    ii. From process entity to sub-process entity dependency
g. Validate dependencies against the model
h. End Sandbox
i. Run the validation process (ProEPV-A.v2) to validate the new process does its intended function

6. Using MICE, the user selects the dependency between Pol-A and ProA-A and opens it in a model editor, where the user changes the dependency to point to the new process ProA-A'.
7. The user updates the view in MICE. MICE uses ProAEI-A and ProEPV-A.v2 to create the graph-view, therefore the list of entities affected by the policy (photos on the website) are now in an invalidated state (red state).
8. Using MICE, the user launches the WE with the process ProA-A' over the invalidated entities. The workflow is run in the sandbox and requires the user to confirm the update.
9. The user updates the view in MICE and checks that now all the entities are in a validated state.
10. The user decides the policy change is correct so he approves it → enact update and commit entities and dependencies in the ERMR.

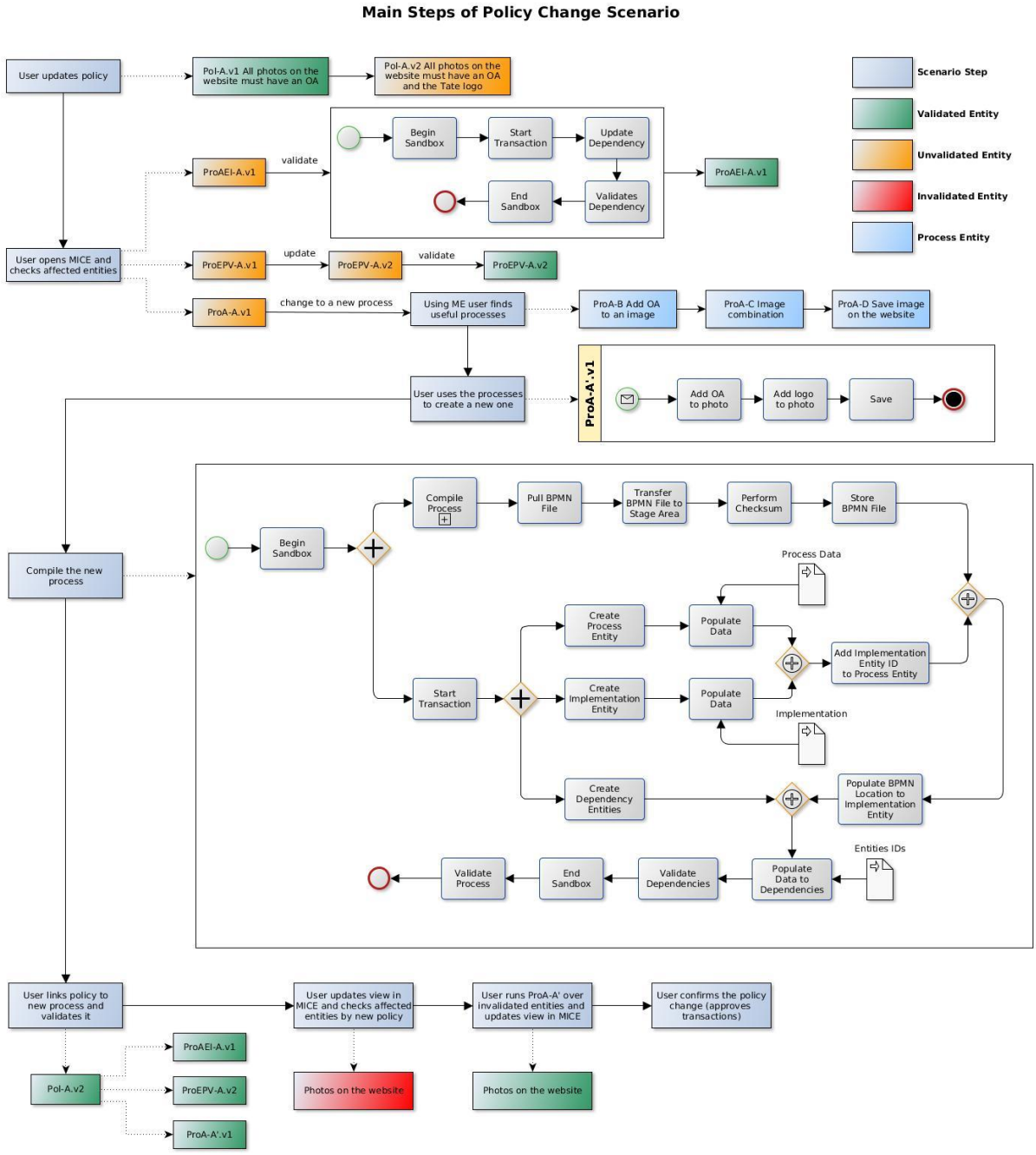

Figure 8-1 Policy Change Scenario Outline Steps

B Appendix – Noise Reduction scenario

B.1 Motivation – space science calibration experiments

The space science case study takes the SOLAR space science experiment as an exemplar. In this experiment, raw observations of the spectrum of the sun are produced using instruments based on the International Space Station (ISS). This results in datasets, which contain sets of observations taken over a period of time. These raw data are analysed and calibrated by a team of scientists using a complex set of scripts, calling mathematical libraries from MATLAB. Calibration is an iterative process running over a number of years. Calibration of the data is carried out in different stages, referred to as levels. These range typically from 0 to 4. At each level, an increasing number of parameters are taken into account. Increased understanding of the behaviour of the instruments and phenomena which may bias the observations (e.g. arrival of vehicles at the ISS) mean that previous calibrations need to be recomputed. The software platforms on which the data gathering and processing are performed are themselves subject to change. For instance, new library versions or algorithms may introduce minor discrepancies into the results. Assessment of the quality of the calibration experiments is typically performed by correlating the results of many experiments, requiring specialist scientific knowledge.

B.2 Description

Noise reduction in images was chosen as a simple proxy for this type of space science experiments. This has several advantages. Noise reduction is well documented and there are many freely available open source software packages. It does not require access to potentially sensitive datasets. Noise can be introduced into images in a controlled and measurable way, and hence we can easily determine quality criteria for the experiments. Thus we are also not reliant on specialised scientific knowledge for testing and development. At the same time, due to the conceptually similar nature of the experiments, and the modular design of the test bed, it should be straightforward to substitute noise reduction with the space science calibration or other experiments. Whilst the data used here is not ordered as a time series, this has little impact on this specific scenario, which has to do with assessment of quality of outcome rather than the specific choice of algorithm.

The aim of the scenario is to investigate two specific issues.

1. Appraisal of experimental runs, where newer calibrations are compared with older results, and some results are marked as less accurate or reliable.
2. Technology change such as updates to software platforms. For example, a new version of a maths library may increase the accuracy of a data analysis operation. It may consequentially be desirable to refactor older calibration results to take advantage of this.

The first point would allow us to eliminate runs of an experiment by checking if they validate a given hypothesis. This might involve providing an algorithm to detect absurd measurements, and potentially what is wrong. In some cases, we may want to appraise intermediate results or do a backwards review of the data to identify the source of an error.

The overall objective of point 2 is to determine if an actual or potential technical change to the experimental environment of a stored experiment may require refactoring (i.e. reprocessing) of the data. This could be done at an individual experiment level or across multiple experiments using

similar platforms. The outcome would first be to determine the risks to the data through certain types of change and the second to determine and implement mitigating actions. This links to the investigations in WP5.

In the basic scenario, an image with added noise is processed by a script calling image processing libraries, and referencing one or more parameter files. The final step is quality measurement, where the effectiveness of the noise reduction processing is determined. The basic scenario can be modified in a number of ways. Noise reduction algorithms can be chained together to emulate the calibration levels in space science. Parameters, algorithms and software libraries used can also be modified. We can also evaluate different quality criteria.

B.3 Requirements

In this scenario, the tasks in a noise reduction process are captured using the BPMN notations within a BPMN process diagram [2]. The BPMN process diagram will show the tasks, their order of execution and any dependencies.

jBPM as described in section 4.3.3 is used to implement the BPMN process.

This scenario involves a number of competing algorithms. For convenience, multiple software libraries are used to implement these algorithms. Concretely, the algorithms compare currently include *n*-iterations of ImageMagick's denoise function and *n*-iterations of the Rudin-Osher-Fatemi (ROF) denoising model. The former is controlled via a Bash script. The latter is implemented in Python using the numpy and scipy libraries, after Solem [16]. The resulting error for each, Peak Signal to Noise Ratio (PSNR) and Mean Square Error (MSE), are calculated using inbuilt functions in the former case, and manually in the latter.

Docker is used to containerise the necessary test data, noise reduction scripts (i.e., the algorithms) and their dependent software. The jBPM process contains a task that invoke the docker container to execute the noise reduction scripts. First a Docker image is created to lay the software infrastructure and data that are necessary for the process. Then one or more containers base on the image is deployed to execute the process.

B.4 Test/Sample Data

Sample images are taken from the SIPI image database [17]. Once the noise reduction script executes the algorithm then produces another image (i.e., an image with less noise) as well as values for Peak Signal to Noise Ratio (PSNR) and/or Mean Square Error (MSE). PSNR and MSE are used for comparison of accuracy of the image processing as described in section B.6. The final image, PSNR, and if reported, MSE values as well as the experiment ID and a reference to the original image are stored in the PERICLES Data Storage for any future comparisons.

B.5 Main Steps

The basic scenario is to run noise reduction on a single Docker container using a single algorithm that requires a particular software stack. This basic process captured using BPMN notation is shown Figure 8-2.

Each execution of the Noise Reduction process is given an Experiment ID. Experiment IDs are necessary, for example, when results for new calibrations are compared with older results so that more accurate (or less accurate) results could be identified and tagged.

The task “Submit Job” runs a jBPM Noise Reduction job using the input values passed into it. The input values include a sample image to be used, script (i.e., the algorithm) to be used, if relevant, number of iterations to be executed within the script and details necessary to access the Data Storage. The Noise Reduction job is a REST service (i.e., accessed via a Handler Wrapper) which uses a Docker container to execute the job.

Once the task is started its “Payload ID” is retrieved from the REST service so that the job can be queried to check whether it is completed or not. The task “Check Status Call” checks the status of the job and then the “Update Status” task updates whether job is finished or not. If the job is not finished then the process check the status of the job again. This repeats until the status is indicated as “finished”.

When the job has finished the results are stored using the “Store Output” task which stores the results to the PERICLES Data Storage and then the process is terminated.

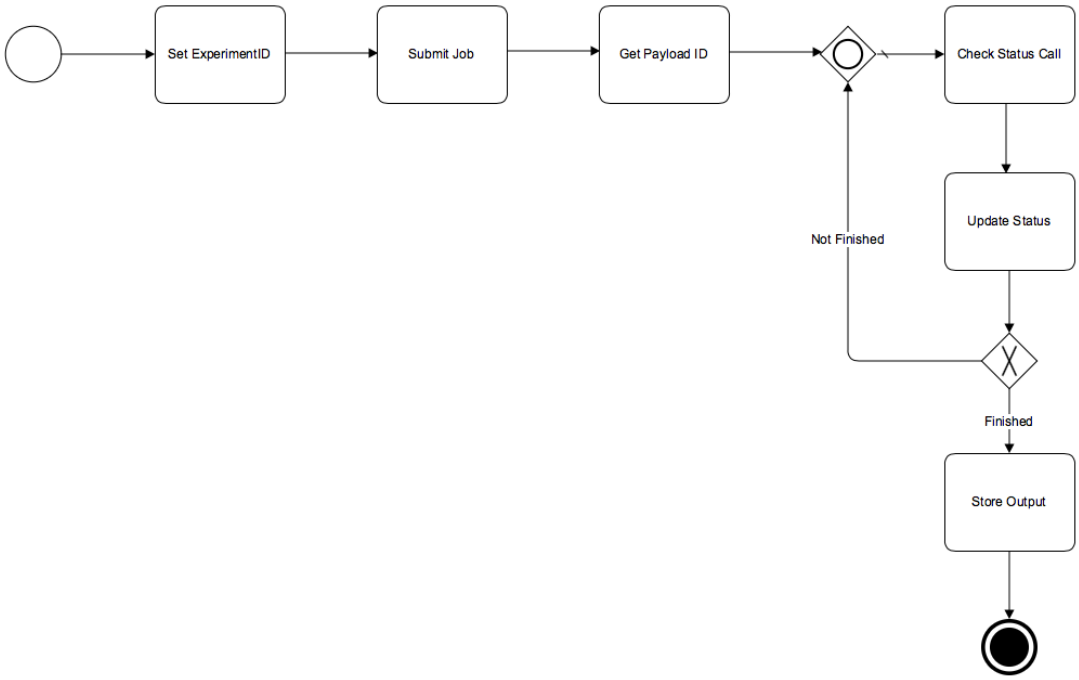

Figure 8-2: BPMN diagram for the Noise Reduction process

The BPMN process shown in Figure 8-2 is implemented using jBPM. The whole process is initiated and executed using the jUnit framework linked via Jenkins. Results of each test runs are shown in the Jenkins dashboard.

B.6 Evaluation of Success Criteria

The success criteria applied are PSNR (peak signal to noise ratio) and MSE (mean square error), which are generally used for this purpose [21] [18]. These are directly mathematically related: concretely, PSNR= 10.log10 [(255)2/MSE] (ibid., equation 2); however, it is conventional to report both.

It is worth remarking that such a measurement is not generally considered sufficient where qualitative views on the quality of image processing are considered. Notably, there are known

limitations to these measurements [20] [19]. In particular, artefacts of image processing may exist that have significant subjective or aesthetic effects on the image and therefore on qualitative evaluation of the quality of a result; in some cases, these may have a low impact on PSNR/MSE measurements, which does not seek to emulate the human visual system. Consequentially, alternative measures of image quality have been proposed [21].

For the purposes of this scenario, these measurements may be taken as sufficiently accurate for our purposes. In particular, the use case we simulate here does not consider subjective or qualitative views on the data. The same assumptions are taken here.

B.7 Extensions to the scenario

The basic process shown in Figure 8-2 is kept generic, with generic names, so that the same process could be used for executing different scenarios. For example, if a different algorithm to be used then this is just change of input to the “Submit Job” task in indicating which script (that encapsulates the algorithm) to be used. If a different software stack to be used, for example different version of Image Magick (image processing library used in the above scenario), then a different service that uses the container with required software stack is used. This separation of process and how it is implemented within tasks allows implementations to be changed without changing the process.

C Appendix – Process Entities

An aggregated process is a process entity that can be described as a combination of other process entities. The arrangement of processes is limited to a flat sequence of sub-processes, understanding as a flat sequence a unique thread with a start event, some processes and a terminate end event, as well as a limited input and output mapping. Input and output resources are represented by files on the file system or arbitrary URIs. These are either provided as input to the new aggregated process or created by a sub-process during execution. A resource can be used as input for any number of sub-processes or as output of the new aggregated process. Resources are immutable, therefore if a sub-process modify an input resource it has to create a new one and of course a resource that is created by a sub-process can only be used by subsequent sub-processes.

On the other hand, an atomic process entity cannot be decomposed in other process entities. Even when its execution may include more than one task from the point of view of the user it represents a single process. Atomic processes will have the information about the operators, the entities that perform the tasks involved in the execution of the workflow.

An aggregated process can be described by a sequence containing atomic and/or other aggregated sub-processes, which can then also described by a combination of atomic or aggregated sub-processes and so on. This aggregation hierarchy follows a tree structure, which always ends in atomic processes with their corresponded operator entities.

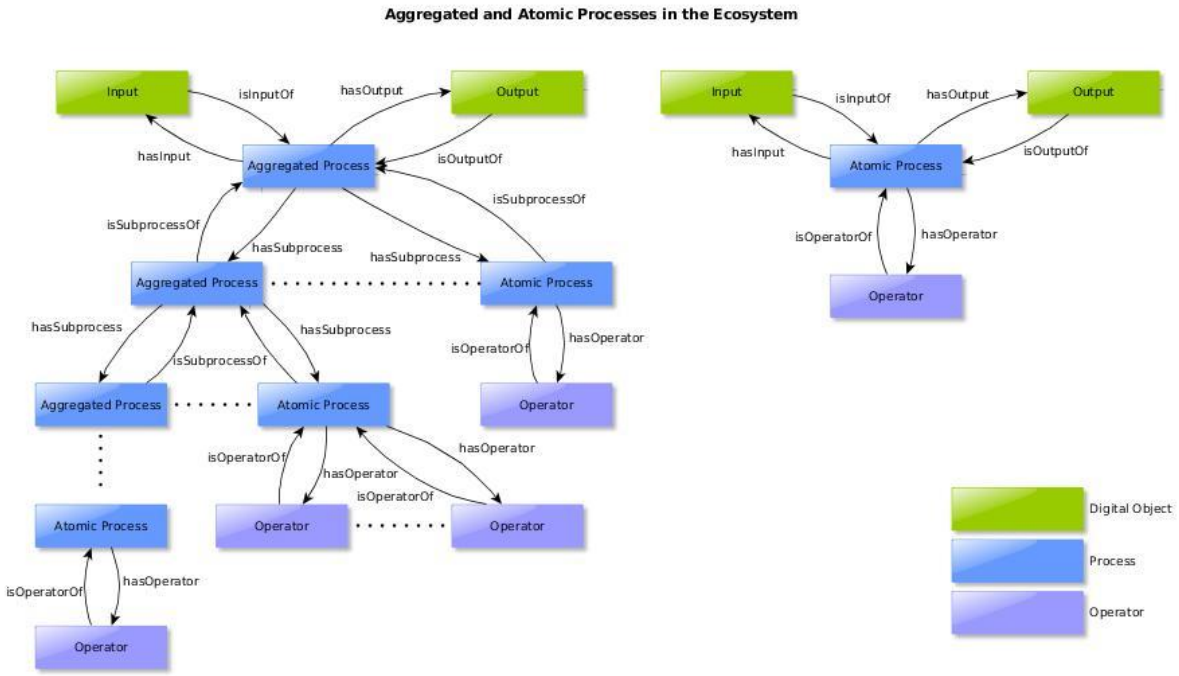

Figure 8-3 Process Entities

C.1 Common Attributes

The common attributes of aggregated and atomic processes are:

- Name: the name of the process (it is not necessary to be unique but human readable), desirably a descriptive one.

- Description: description of the process, its purpose, which operation/functionality it performs, over which entities, in which conditions, etc.
- Identity: a unique identifier (i.e. URI).
- Version: a version number.
- Input: list of input slots *(0..n), specifying for each one
    - Name: variable name, a human readable name of the input slot.
    - Description: description of the input, its functionality and flow during the process.
    - Identity: a unique identifier (i.e. URI) of the slot (to avoid misunderstanding when combining processes with same variable names).
    - Input Type: type of entities allowed as input (lower entities in the hierarchy are also allowed).
    - Optional: a boolean flag to specify if the input is required or optional.
- Output: list of output slots *(0..n), specifying for each one
    - Name: variable name, a human readable name of the output slot.
    - Description: description of the output, its functionality and flow during the process.
    - Identity: a unique identifier (i.e. URI) of the slot (to avoid misunderstanding when combining processes with same variable names).
    - Output Type: type of entities allowed as output (lower entities in the hierarchy are also allowed).
- Implementation: the unique identifier (i.e. URI) of the entity containing the information about the implementation (implementation type, i.e. BPMN, file location, checksum, etc. see Implementation Entity description below).

C.2 Specific Attributes of Atomic Processes

The specific attribute of an atomic process entity is:

- Operator: list of operators *(0..n), those entities responsible for performing the tasks in the workflow.

C.3 Specific Attributes of Aggregated Processes

The specific attribute of an aggregated process entity is:

- Sequence: list of sub-processes, specifying for each one
    - Process Identifier: the unique identifier (i.e. URI) of the process.
    - Input Map: mapping between available resources at that step and the input slots of the process.
    - Output Map: mapping between the output slots of the process and new temporary resources.

As only flat sequences are allowed, that is a serial execution of the processes without gateways or loops where the start event corresponds to the start event of the first sub-process and the end-event corresponds to the end-event of the last sub-process, the control flow in the sequence is determined by the list of process identifiers. The reason behind this constrain is to make easier the reasoning during compilation. Notice that is possible to have the same process more than once in the sequence. The data flow is determined by the input and output mapping. Therefore, each step of the sequence represents the execution of a sub-process with a well-defined data set.

C.4 Implementation Entity

An implementation entity contains the information about the file with the low-level description of a process. Its attributes are the following:

- Identity: a unique identifier (i.e. URI)
- Version: a version number
- Type: type of implementation, i.e., BPMN description
- Location: the location to the implementation file (the low-level description of the process workflow, e.g. a BPMN file)
- Fixity: a unique identifier (i.e. URI) of a fixity entity to validate data integrity
    - Checksum: the checksum string
    - Algorithm: the algorithm used to calculate the checksum

D Appendix – Handlers Technical Information

D.1 Error Codes

Code	Description
10	Error in scripts running before payload's command (defined in config)
20	Error in payload's command execution
30	Error in scripts running after payload's command (defined in config)
40	Error in Source data fetch
50	Error in Destination data creation
60	Store/Queue error
100	Unknown error

Table 7 Processing Error Codes

D.2 Handler Status Codes

Code	Status
0	Dead
1	Idle
2	Working
3	Error

Table 8 Handler Status Codes

D.3 Handler Configuration

key	description
path	Path where component binaries are stored.
hostname	handler's default hostname (if not defined in cli)
port	default port (if not defined in cli), defaults to 8008
log_file	log file path

commands	commands that handler supports, command name is they key
commands[].params[]]	parameters for current command, parameter name is the key
commands[].params[].under	sign between param name and value, e.g. "=", defaults to ""
commands[].params[].required	true/false
commands[].params[].default	if required, define default value if exists
commands.flags	command flags list, e.g. [-l, -m]
commands.script	"${CMP_PATH}/{script_file_name}" script path
commands.commit	message on commit processed data
commands.before_script	shell scripts executed before command
commands.after_script	shell scripts called after command
transfers	transfer methods config
transfers.repo	
transfers.repo.local_dir	working dir
transfers.repo.repos_dir	dir storing the processed data repo
transfers.repo.repos_uri	url providing repositories (after processing)
transfers.http	
transfers.http.local_dir	working dir
transfers.http.files_dir	processed files dir
transfers.http.files_url	url providing processed data

Table 9 Handler Configuration Elements

D.4 Endpoint Summary Tables

Key	**Description**
status.code	Handler status code
status.msg	Detailed message of status

Table 10 Handler Status Values

Key	Description
cmd	command to be executed
params	key-value list of command's parameters
flags	list of flags to be added to command call
source.type	format of data to be processed source (http/repo)
source.url	format of data to be processed url (url)
destination.type	format of data to be returned source (http/repo)

Table 11 Payload Objects Definitions

Type	Description
repository	Source URL refers to a git repository. The Handler clones the repository to the local working directory for command execution.
http	Source URL is an HTTP link to a compressed data file (zip, gzip, tar, targz). The Handler downloads this archive, decompresses it to the working directory and executes the commands. When finished, the processed data is compressed and a username/password authenticated URL HTTP address is published contained the resultant archive file.

Table 12 URL Types

E Appendix – ERMR Technical Information

E.1 Create New Model

Header	Type	Description	Requirement
Accept	Header String	“application/cdmi-object”	Optional
Content-Type	Header String	“application/cdmi-object”	Mandatory
X-CDMI-Specification-Version	Header String	“1.1”	Mandatory

Table 13 Request Headers

Field Name	Type	Description	Requirement
mimetype	JSON String	Mime type of the data contained within the value field	Optional
metadata	JSON Object	Metadata for the data object	Optional
value	JSON String	The data object value	Optional

Table 14 Request Message Body

Header	Type	Description	Requirement
Content-Type	Header String	“application/cdmi-object”	Mandatory
X-CDMI-Specification-Version	Header String	“1.1”	Mandatory

Table 15 Response Headers

Field Name	Type	Description	Requirement

objectType	JSON String	“application/cdmi-object”	Mandatory
objectID	JSON String	ObjectID of the object	Mandatory
objectName	JSON String	Name of the object	Mandatory
parentURI	JSON String	URI for the parent object	Mandatory
parentID	JSON String	Object ID of the parent object	Mandatory
mimetype	JSON String	MIME type of the value of the data object	Mandatory
metadata	JSON Object	Metadata for the object	Mandatory

Table 16 Response Message Body

HTTP Status	Description
201 Created	The new data object was created
400 Bad Request	The request contains invalid parameters.
401 Unauthorized	The authentication credentials are missing or invalid.
403 Forbidden	The client lacks the proper authorization
404 Not Found	The resource was not found at the specified URI

Table 17 Response Status

E.2 Get Model

Header	Type	Description	Requirement
Accept	Header String	“application/cdmi-object”	Optional
X-CDMI-Specification-Version	Header String	“1.1”	Mandatory

Table 18 Request Headers

Header	Type	Description	Requirement
Content-Type	Header String	“application/cdmi-object”	Mandatory
X-CDMI-Specification-Version	Header String	“1.1”	Mandatory

Table 19 Response Headers

Field Name	Type	Description	Requirement
objectType	JSON String	“application/cdmi-object”	Mandatory
objectID	JSON String	ObjectID of the object	Mandatory
objectName	JSON String	Name of the object	Mandatory
parentURI	JSON String	URI for the parent object	Mandatory
parentID	JSON String	Object ID of the parent object	Mandatory
mimetype	JSON String	MIME type of the value of the data object	Mandatory
metadata	JSON Object	Metadata for the object	Mandatory
value	JSON String	data object value	Conditional

Table 20 Response Message Body

HTTP Status	Description
20O OK	The data object content was returned in the response.
401 Unauthorized	The authentication credentials are missing or invalid
403 Forbidden	The client lacks the proper authorization
404 Not Found	The resource was not found at the specified URI

Table 21 Response Status